Franck Cappello
Thomas Herault
Jack Dongarra (Eds.)

LNCS 4757

Recent Advances in Parallel Virtual Machine and Message Passing Interface

14th European PVM/MPI Users' Group Meeting
Paris, France, September/October 2007
Proceedings

Lecture Notes in Computer Science 4757

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Franck Cappello Thomas Herault
Jack Dongarra (Eds.)

Recent Advances in Parallel Virtual Machine and Message Passing Interface

14th European PVM/MPI Users' Group Meeting
Paris, France, September 30 - October 3, 2007
Proceedings

Springer

Volume Editors

Franck Cappello
INRIA Futurs
LRI, Bat 490 University Paris South
91405 Orsay France
E-mail: franck.cappello@lri.fr

Thomas Herault
Universite Paris Sud-XI
Laboratoire de Recherche en Informatique
Bâtiment 490
91405 Orsay France
E-mail: thomas.herault@lri.fr

Jack Dongarra
University of Tennessee
Computer Science Department
1122 Volunteer Blvd, Knoxville, TN 37996-3450, USA
E-mail: dongarra@cs.utk.edu

Library of Congress Control Number: 2007935601

CR Subject Classification (1998): D.1.3, D.2.3, F.1.2, G.1.0, B.2.1, B.2.1, C.1.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-75415-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-75415-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12168712 06/3180 5 4 3 2 1 0

Preface

Thirteen years after the publication of the first MPI (message passing interface) specification and 17 years after the first published paper on PVM (parallel virtual machine), MPI and PVM have emerged as standard programming environments and continue to be the development environment of choice for a large variety of applications, hardware platforms, and usage scenarios. There are many reasons behind this success and one of them is certainly the strength of its community.

EuroPVM/MPI is a flagship conference for this community, established as the premier international forum for researchers, users, and vendors to present their latest advances in MPI and PVM. EuroPVM/MPI is the forum where fundamental aspects of message passing, implementations, standards, benchmarking, performance, and new techniques are presented and discussed by researchers, developers and users from academia and industry.

EuroPVM/MPI 2007 was organized by INRIA in Paris, September 29 to October 3, 2007. This was the 14th issue of the conference, which takes place each year at a different European location. Previous meetings were held in Bonn (2006), Sorrento (2005), Budapest (2004), Venice (2003), Linz (2002), Santorini (2001), Balatonfüred (2000), Barcelona (1999), Liverpool (1998), Krakow (1997), Munich (1996), Lyon (1995), and Rome (1994).

The main topics of the meeting were collective operations, one-sided communication, parallel applications using the message passing paradigm, MPI standard extensions or evolution, fault tolerance, formal verification of MPI processes, MPI-I/O, performance evaluation, and hierarchical infrastructures (parallel computers and Grids).

For this year's conference, the Program Committee Co-chairs invited six outstanding researchers to present lectures on different aspects of the message passing paradigm: Tony Hey, who co-authored the first draft for the MPI standard, presented "MPI: Past, Present and Future," Al Geist, one of the authors of PVM, presented "Sustained PetaScale, the Next MPI Challenge," Satoshi Matsuoka, a pioneer of the Grid Computing, presented "The Tsubame Cluster Experience," Ewing Lusk, one of the leaders of MPICH, presented "New and Old Tools and Programming Models for High-Performance Computing," George Bosilca, one of the leading members of OpenMPI, presented "The X-Scale Challenge," and Bernd Mohr, a pioneer in performance analysis tools for Parallel Computing, presented "To Infinity and Beyond!?".

In addition to the conference main track, the meeting featured the sixth edition of the special session "ParSim 2007 – Current Trends in Numerical Simulation for Parallel Engineering Environments." The conference also included three tutorials, one on "Using MPI-2: A Problem-Based Approach" by William Gropp and Ewing Lusk, the second by Stephen Siegel on "Verifying Parallel Programs with MPI-Spin," and the third by George Bosilca and Julien Langou on "Advanced MPI Programming."

Contributions to EuroPVM/MPI 2007 and the special session ParSim were submitted in May and June, respectively. Out of the 68 submitted full papers, 40 were selected for presentation at the conference. The task of reviewing was carried out smoothly within very strict time limits by a large Program Committee, which included members from most of the American and European groups involved in MPI and PVM development, as well as from significant user communities. Almost all papers received four reviews, some even five, and none fewer than three, which provided a solid basis for the Program Chairs to make the final selection for the conference program. The result was a well-balanced, focused, and high-quality program. Out of the accepted 40 papers, four were selected as outstanding contributions to EuroPVM/MPI 2007, and were presented in special, plenary sessions:

- "Full Bandwidth Broadcast, Reduction and Scan With Only Two Trees" by Peter Sanders, Jochen Speck and Jesper Larsson Traff
- "Process Cooperation in Multiple Message Broadcast" by Bin Jia
- "Self-Consistent MPI Performance Requirements" by Jesper Larsson Traff, William Gropp and Rajeev Thakur
- "Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE" by Rajeev Thakur and William Gropp

An important part of EuroPVM/MPI is the technically oriented vendor session. At EuroPVM/MPI 2007 seven significant vendors of hardware and software for high-performance computing (Microsoft, Hewlett Packard, IBM, CISCO, Myricom, Intel, and Voltaire) presented their latest products and developments.

Information about the conference can be found at the conference Web site: http://pvmmpi07.lri.fr/, which will be kept available.

The proceedings were edited by Franck Cappello and Thomas Herault. The EuroPVM/MPI 2007 logo was designed by Ala Rezmerita.

The program and general chairs would like to thank all who contributed to making EuroPVM/MPI 2007 a fruitful and stimulating meeting, be they technical paper or poster authors, Program Committee members, external referees, participants, or sponsors. We would like to express our gratitude to all the members of the Program Committee and the additional reviewers, who ensured the high quality of Euro PVM/MPI 2007 with their careful work.

Finally, we would like to thank deeply INRIA and LRI for their support and efforts in organizing this event. In particular, we would like to thank Catherine Girard (INRIA), Nicole Lefevre (LRI), Gaelle Dorkeld (INRIA), and Chantal Girodon (INRIA). A special thanks to all PhD students who helped in the logistics of the conference.

October 2007

Franck Cappello
Thomas Herault
Jack Dongarra

Organization

General Chair

Jack J. Dongarra	University of Tennessee, Knoxville, USA

Program Chairs

Franck Cappello	INRIA
Thomas Herault	Université Paris Sud-XI / INRIA

Program Committee

George Almasi	IBM, USA
Ranieri Baraglia	CNUCE Institute, Italy
Richard Barrett	ORNL, USA
Gil Bloch	Mellanox, Israel
George Bosilca	Univeristy of Tennesse, USA
Hakon Bugge	Scali, Norway
Franck Cappello	University of Paris-Sud, France
Barbara Chapman	University of Houston, USA
Brian Coghlan	Trinity College Dublin, Ireland
Yiannis Cotronis	University of Athens, Greece
Jose Cunha	New University of Lisbon, Portugal
Marco Danelutto	University of Pisa, Italy
Luiz DeRose	Cray, USA
Frederic Desprez	INRIA, France
Erik D'Hollander	University of Ghent, Belgium
Beniamino Di Martino	Second University of Naples, Italy
Jack Dongarra	University of Tennessee, USA
Edgar Gabriel	University of Houston, USA
Al Geist	OakRidge National Laboratory, USA
Patrick Geoffray	Myricom, USA
Michael Gerndt	Technical University of Munich, Germany
Andrzej Goscinski	Deakin University, Australia
Richard L. Graham	ORNL, USA
William Gropp	Argonne National Laboratory, USA
Rolf Hempel	DLR - German Aerospace Center, Germany

Thomas Herault	Université Paris Sud / INRIA, France
Yutaka Ishikawa	University of Tokyo, Japan
Rainer Keller	HLRS, Germany
Stefan Lankes	RWTH Aachen, Germany
Erwin Laure	CERN, Switzerland
Laurent Lefevre	INRIA, France
Greg Lindahl	Pathscale, USA
Thomas Ludwig	University of Heidelberg, Germany
Ewing Rusty Lusk	Argonne National Laboratory, USA
Tomas Margalef	Universitat Autonoma de Barcelona, Spain
Jean-Franois Mhaut	IMAG, France
Bernd Mohr	Forschungszentrum Jülich, Germany
Matthias Müller	Dresden University of Technology, Germany
Raymond Namyst	University of Bordeaux, France
Salvatore Orlando	University of Venice, Italy
Christian Perez	IRISA, France
Fabrizio Petrini	PNNL, USA
Neil Pundit	Sandia National Laboratories, USA
Rolf Rabenseifner	HLRS, Germany
Thomas Rauber	Universität Bayreuth, Germany
Casiano Rodriguez-Leon	University of La Laguna, Spain
Martin Schulz	Lawrence Livermore National Laboratory, USA
Jeffrey Squyres	Cisco, Inc., USA
Bernard Tourancheau	University of Lyon / INRIA, France
Jesper Larsson Träff	C&C Research Labs, NEC Europe, Germany
Carsten Trinitis	Technische Universität München, Germany
Roland Wismueller	University Siegen, Germany
Felix Wolf	Forschungszentrum Jülich, Germany
Joachim Worringen	C&C Research Labs, NEC Europe, Germany

External Referees

Toni Cortes	Universitat Politècnica de Catalunya, Spain
Pierre Lemarinier	Université Paris Sud/INRIA, France

Conference Organization

Franck Cappello	INRIA
Gaelle Dorkeld	INRIA
Catherine Girard	INRIA
Chantal Girodon	INRIA
Thomas Herault	Université Paris Sud-XI / INRIA

Sponsors

The conference would have been significantly more expensive and much less pleasant to organize without the generous support of our industrial sponsors. Platinum and Gold level sponsors also gave talks at the vendor session on their latest products in parallel systems and message passing softwares. EuroPVM/MPI 2007 gratefully acknowledges the contributions of the sponsors to a successful conference.

Platinum Level Sponsors

Gold Level Sponsors

Standard Level Sponsors

Table of Contents

Invited Talks

Tutorials

Outstanding Papers

Applications

Collective Operations

Fault Tolerance

Library Internals

Message Passing on Hierarchical Machines and Grids

MPI-I/O

One-Sided

PVM and Harness

Tools

Verification of Message Passing Programs

ParSim

Posters Abstracts

The X-Scale Challenge

George Bosilca

Innovative Computing Laboratory
Electrical Engineering and Computer Science Department,
University of Tennessee, Knoxville, TN, USA
bosilca@cs.utk.edu

The last few years have seen fast changes in the high performance computing field. Some of these changes are related to the computer hardware and architecture. Multi/many core architecture have become prevalent, with architectures more or less exotic and heterogeneous. The overall theoretical computational power of the new generation processors increased greatly, but their programmability still lacks confidence. The changes in shape of the newest architectures has come so rapidly, that we are still deficient in terms of high performance libraries and applications in order to take advantage of all these new features.

At same time, application requirements grow at least at the same pace. Obviously, more computations require more data in order to feed the deepest processor pipelines. More data means either a faster access to the memory, or a faster access to the network. The access speed to all types of memory (network included) lags behind. As a result, extracting the right performance of the current and next generation architectures is still (and will remain for a while) a challenge. These two fields are coming along, providing some very interesting advances over the last few years, but not at the speed of FLOPS. Moreover, the current approach indicates a higher degree of memory hierarchies (Non Uniform Memory Accesses) that have already become another limiting factor for the application performance increase.

Simultaneously, increasing the size of the parallel machines triggers an increase in fault tolerance requirements. While the fault management and recovery topic was thoughtfully studied over the last decade, the recent changes in the number and distribution of the processor's cores have raised some interesting questions. While the question of which fault tolerant approach fits best to the peta-scale environments is still debated, few of these approaches show interesting performances at scale or a low degree of intrusion in the application code. Eventually, the right answer might be somewhere in between a dynamic combination of several of these methods, strictly based on the application's properties and the hardware environment.

As expected, all these changes guarantee a highly dynamic (and exciting from a research point of view) high performance field over the next years. New mathematical algorithms will have to emerge in order to take advantage of these unbalanced architectures. In addition, a tight collaboration between the applications and the high performance libraries developers is/will be *critical* to the future of HPC fields.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 1–2, 2007.

How MPI will adapt to fit into this conflicting environment is still an open question. Over the last few years, MPI has been a very successful parallel programming paradigm, partially due to its apparent simplicity to express basic message exchange patterns and partially to the fact that it increases the productivity of the programmers and the parallel machines. Whatever the future of MPI will be, these two features should stay an indispensable part of its new direction of development.

Sustained Petascale: The Next MPI Challenge

Al Geist

Oak Ridge National Laboratory,
PO Box 2008,
Oak Ridge, TN 37831-6016
gst@ornl.gov
http://www.csm.ornl.gov/ geist

Abstract. The National Science Foundation in the USA has launched an ambitious project to have a sustained petaflop computer in production by 2011. For applications to run at a sustained petaflop, the computer will need to have a peak performance of nearly 20 PF and millions of threads. This talk will address the challenges MPI must face to be used in sustained petascale applications.

The first significant challenge for MPI will be the radical change in supercomputer architectures over the next few years. The architectures are shifting from using faster processors to using multi-core processors. It is also speculated that the processors will be heterogeneous with the cores on a single processor having different functions. This change in architecture is as disruptive to software as the shift from vector to distributed memory supercomputers 15 years ago. That change required complete restructuring of scientific application codes and gave rise to the message passing programming paradigm that drove the popularity of PVM and MPI. Similarly, will these new architectures drive the creation of a new programming paradigm or will MPI survive? Or perhaps MPI becomes part of a new hybrid paradigm. These questions will be addressed in this talk.

The configuration of these sustained petascale systems will require applications to exploit million-way parallelism and significant reductions in the bandwidth and amount of memory available to a million cores. Most science teams have no idea how to efficiently scale their applications to this level and those teams that have thought about it believe that MPI may not be the right approach. Several talks at this conference describe potential features and capabilities that may allow MPI to be more effective for the reduced bandwidth and increased parallelism. The talk will point out these features and their associated talks.

The third significant challenge for MPI will be fault tolerance and fault recovery. The amount of memory in sustained petascale systems makes writing out checkpoints impractical. While the need to restart an application when 900,000 CPUs are still working fine but one has failed is an inefficient use of resources. This talk will describe the latest ideas for fault recovery in MPI and will describe a new idea called holistic fault tolerance that is being investigated.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 3–4, 2007.

Finally the talk will describe productivity issues for sustained petascale application performance and their implications for MPI. The productivity of scientists and engineers is based on how easy and how fast they can solve a new science problem. Issues such as debugging, performance tuning, scalability, validation, and knowledge discovery all play a part. The talk will address the challenges MPI has in these areas.

MPI: Past, Present and Future

Tony Hey

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Abstract. This talk will trace the origins of MPI from the early message-passing, distributed memory, parallel computers in the 1980's, to today's parallel supercomputers. In these early days, parallel computing companies implemented proprietary message-passing libraries to support distributed memory parallel programs. At the time, there was also great instability in the market and parallel computing companies could come and go, with their demise taking with them a great deal of effort in parallel programs written to their specific call specifications. In January 1992, Geoffrey Fox and Ken Kennedy had initiated a community effort called Fortran D, a precursor to High Performance Fortran and a high level data parallel language that compiled down to a distributed memory architecture. In the event, HPF proved an over ambitious goal: what was clearly achievable was a message-passing standard that enabled portability across a variety of distributed memory message-passing machines. In Europe, there was enthusiasm for PARMACS libraries: in the US, PVM was gaining adherents for distributed computing. For these reasons, in November 1992 Jack Dongarra and David Walker from the USA and Rolf Hempel and Tony Hey from Europe wrote an initial draft of the MPI standard. After a birds of a feather session at the 1992 Supercomputing Conference, Bill Gropp and Rusty Lusk from Argonne volunteered to create an open source implementation of the emerging MPI standard. This proved crucial in accelerating take up of the community-based standard, as did support from IBM, Intel and Meiko. Because of the need for the MPI standardization process to converge to agreement in a little over a year, the final agreed version of MPI contains more communication calls than most users now require. A later standardization process increased the functionally of MPI as MPI-2.

Where are we now? It is clear that MPI provides effective portability for data parallel distributed memory message passing programs. Moreover, such MPI programs can scale to large numbers of processors. MPI therefore still retains its appeal for closely coupled distributed computing and the rise of HPC clusters as a local resource has made MPI ubiquitous for serious parallel programmers. However, there are two trends that may limit the usefulness of MPI in the future. The first is the rise of the Web and of Web Services as a paradigm for distributed, service oriented computing. In principle, using Web Service protocols that expose functionality as a service offers the prospect of building more robust software for distributed systems. The second trend is the move towards Multi-Core processors as semi-conductor manufacturers are finding that they

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 5–6, 2007.

can no longer increase the clock speed as the feature size continues to shrink. Thus, although Moore's Law, in the sense that the feature size will continue to shrink, will continue for perhaps a decade or more, the accompanying increase in speed as the clock speed is increased will no longer be possible. For this reason, 2, 4 and 8 core chips, in which the processor is replicated several times, are already becoming commonplace. However, this means that any significant performance increase will depend entirely on the programmer's ability to exploit parallelism in their applications. This talk will end by reviewing these trends and examining the applicability of MPI in the future.

New and Old Tools and Programming Models for High-Performance Computing

Ewing Lusk

Mathematics and Computer Science Division
Argonne National Laboratory

Abstract. The computing power to be made available to applications in the coming years continues to increase. Hardware vendors anticipate many cores on single chips and fast networks connecting them, enabling a bewildering array of new approaches to parallel programming whose superiority to "classical" approaches (MPI) remains uncertain. One certainty is that application developers will need tools that promote *understanding* of their parallel codes. In this talk we will review a number of approaches to application development, from Fortran77+MPI to the relatively exotic DARPA "high productivity" languages, together with a sampling of tools, both new and old, that aid programmers in application development.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, p. 7, 2007.

The TSUBAME Cluster Experience a Year Later, and onto Petascale TSUBAME 2.0

Satoshi Matsuoka

Global Scientific Information and Computing Center
Tokyo Institute of Technology
2-12-1 Oo-Okayama, Meguro-ku, Tokyo 152-8550, Japan
matsu@is.titech.ac.jp
http://www.gsic.titech.ac.jp

TSUBAME (Tokyo-tech Supercomputer and Ubiquitously Accessible Mass-storage Environment) is a new supercomputer installed at Tokyo Institute of Technology in Tokyo, Japan, on April 1st, 2006, and as of 2007 facilitating over 85 Teraflops of peak compute power with acceleration, 22 Terabytes of memory, and 1.6 Petabytes of online disk storage, "Fat Node" as well as fast parallel interconnect—architectural principles based on traditional supercomputers. TSUBAME became the fastest and largest supercomputer in Asia in terms of performance, memory and storage capacity etc., starting from the 38.18 Teraflops performance (7th overall) for the June 2006 Top500 announcement, and taking the fastest in Asia crown for 3 consecutive Top500s in a row (currently at 48.88 Teraflops). At the same time, being PC architecture-based, TSUBAME, being a large collection of PC servers, allows for offering much broader services than traditional supercomputers resulting in a much wider user base, including incubation of novice students. We term such architectural and operational property of TSUBAME as "Everybody's Supercomputer", as opposed to traditional supercomputers with very limited number of users, thus making their financial justifications increasingly difficult.

Tsubame is a result of collaborative effort between the univerity academia at the Tokyo Institute of Technology, and multiple industrial partners worldwide. The contract was awarded to NEC, who jointly with Sun Microsystems built and installed the entire machine, and also collaboratively provide on-site engineering to operate the machine. Other commercial partners, such as AMD (Opteron CPUs), Voltaire (Infiniband), ClearSpeed (Accelerator), CFS (LUSTRE parallel filesystem), Novell (SUSE Linux), provided their own products and expertise as building blocks. It was installed in just three weeks, and when its operation started on April 3rd, 2006

Overall, TSUBAME's installation space is approximately $350m^2$ including the service area. There are approximately 80 compute/storage/network racks, as well as 32 CRC units for cooling, that are laid out in a customized fashion to maximize cooling efficiency, instead of the machine itself merely being placed as an afterthought. This allows for considerable density and much better cooling efficiency compared to other machines of similar performance. TSUBAME occupies three rooms, where room-to-room Infiniband connections are achieved

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 8–9, 2007.

via optical fiber connection, whereas uses the CX4 copper cable within a room. The total power consumption of TSUBAME is less than a Megawatt even at peak load, making it one of the most power- and space- efficient genera-purpose cluster supercomputer in the 100Teraflops performance scale.

TSUBAME's lifetime was initially designed to be 4 years, until the spring of 2010, with possible short-term extensions realized by incremental upgrades to maintain the competitiveness of the machine. However, eventually the lifetime will expire, and we are already beginning the plans for designing the next generaion "TSUBAME 2.0". Here, simply waiting for processor improvements relying on CPU vendors would not be sufficient to meet the growing computational demands, as a result of success of "Everybody's Supercomputer", in growth of the supercomputing community itself, not just the individual needs. Another requirement is not to increase the power or the footprint requirement of the current machine, resulting in a considerable challenge in supercomputer design we are researching at the current moment.

One research investment we are conducting in this regard is in the area of acceleration technologies, which will provide vastly improved Megaflops/Watt ratio. In fact, even currently, two-fifth of TSUBAME's peak computing power is provided by the ClearSpeed Advanced Accelerator PCI-X board. However, acceleration technology is still narrowly scoped in terms of its applicability and user base; as such, we must generalize the use of acceleration via advances in algorithm and software technologies, as well as design a machine with right mix of various heterogeneous resources, including general-purpose processors, and various types of accelerators. Another factor is storage, where multi-Petabyte storage with high bandwidth must be accommodated. Challenges are in devising more efficient cooling, better power control, etc. There are various challenges abound, and it will require advances in multi-disciplinary fashion to meet this challenge. This is not a mere pursuit of FLOPS, but rather, "pursuit of FLOPS usable by everyone"—a challenge worthwhile taking for those of us who are computer scientists. And the challenge will continue beyond TSUBAME 2.0 for many years to come.

To Infinity and Beyond?! On Scaling Performance Measurement and Analysis Tools for Parallel Programming

Bernd Mohr

Forschungszentrum Jülich
John-von-Neumann Institute for Computing
Virtual Institute for High-Productivity Supercomputing
Jülich, Germany
b.mohr@fz-juelich.de

Extended Abstract. The number of processor cores available in high-performance computing systems is steadily increasing. A major factor is the current trend to use multi-core and many-core processor chip architectures. In the latest list of the TOP500 Supercomputer Sites[1], 63% of the systems listed have more than 1024 processor cores and the average is about 2400.

While this promises ever more compute power and memory capacity to tackle today's complex simulation problems, it forces application developers to greatly enhance the scalability of their codes to be able to exploit it. This often requires new algorithms, methods or parallelization schemes to be developed as many well-known and accepted techniques stop working at these large scales. It starts with simple things like opening a file per process to save checkpoint information, or collecting simulation results of the whole program via a gather operation on a single process, or previously unimportant order $O(n^2)$-type operations which quickly dominate the execution. Unfortunately many of these performance problems only show up when executing with very high numbers of processes and cannot be easily diagnosed or predicted from measurements at lower numbers. Detecting and diagnosing these performance and scalability bottlenecks requires sophisticated performance instrumentation, measurement and analysis tools. Simple tools typically scale very well but the information they provide proves to be less and less useful at these high scales.

It is clear that tool developers face exactly the same problems as application developers when enhancing their tools to handle and support highly scalable applications. In this talk we discuss the major limitations of currently used state-of-the-art performance measurement, analysis and visualisation methods and tools. We give an overview about experiments, new approaches and first results of performance tool projects which try to overcome these limits. This includes new scalable and enhanced result visualization methods used in the performance analysis framework TAU[2], methods to automatically extract key execution phases from long traces used by the Paraver toolset[3], more scalable client/server tool architecture like the one of VampirServer[4] for scalable timeline visualisations, and highly-parallel automatic performance bottleneck searches utilized by the Scalasca toolset[5].

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 10–11, 2007.

References

1. TOP500 Supercomputer Sites (June 2007) → http://www.top500.org/
2. Shende, S., Malony, A.D.: TAU: The TAU Parallel Performance System. International Journal of High Performance Computing Applications 20(2), 287–331 (2006)
3. Labarta, J., Gimenez, J., Martinez, E., Gonzales, P., Servat, H., Llort, G., Aguilar, X.: Scalability of visualization and tracing tools. In: Proceedings Parallel Computing (ParCo) 2005, Malaga, Spain (2005)
4. Knüpfer, A., Brunst, H., Nagel, W.E.: High Performance Trace Visualization. In: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing, Lugano, Switzerland (February 2005)
5. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable Parallel Trace-Based Performance Analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)

Using MPI-2: A Problem-Based Approach

William D. Gropp and Ewing Lusk

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{thakur,gropp}@mcs.anl.gov

Abstract. MPI-2 introduced many new capabilities, including dynamic process management, one-sided communication, and parallel I/O. Implementations of these features are becoming widespread. This tutorial shows how to use these features by showing all of the steps involved in designing, coding, and tuning solutions to specific problems. The problems are chosen for their practical use in applications as well as for their ability to illustrate specific MPI-2 topics. Complete examples that illustrate the use of MPI one-sided communication and MPI parallel I/O will be discussed and full source code will be made available to the attendees. Each example will include a hands-on lab session; these sessions will also introduce the use of performance and correctness debugging tools that are available for the MPI environment. Guidance on tuning MPI programs will be included, with examples and data from MPI implementations on a variety of parallel systems, including Sun, IBM, SGI, and clusters. Examples in C, Fortran, and C++ will be included. Familiarity with basic MPI usage will be assumed.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, p. 12, 2007.

Verifying Parallel Programs with MPI-Spin

Stephen F. Siegel*

Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA
siegel@cis.udel.edu
http://www.cis.udel.edu/~siegel

Standard testing and debugging techniques are notoriously ineffective when applied to parallel programs, due to the numerous sources of nondeterminism arising from parallelism. MPI-Spin, an extension of the model checker Spin for verifying and debugging MPI-based parallel programs, overcomes many of the limitations associated with the standard techniques. By exploring *all possible executions* of an MPI program, MPI-Spin can conclude, for example, that a program cannot deadlock on any execution. If the program can deadlock, MPI-Spin can exhibit a *trace* showing exactly how the program fails, greatly facilitating debugging.

This tutorial will serve as an introduction to MPI-Spin. Through a series of examples and exercises, participants will learn to use MPI-Spin to check for deadlocks, race conditions, and discrepancies in the numerical computations performed by MPI programs. The only prerequisites are familiarity with C and the basic MPI operations; no prior verification experience is required. Participants are encouraged to download and install MPI-Spin before the tutorial, following the instructions at http://vsl.cis.udel.edu/mpi-spin.

The tutorial is divided into four parts, each lasting approximately 45 minutes: (1) introduction and tool demonstration, (2) language basics, (3) using MPI-Spin, and (4) verifying correctness of numerical computations.

1. Introduction and Demonstration. The introduction will begin with a discussion of some of the most common problems plaguing developers of MPI programs. In addition to the issues mentioned above, issues related to performance, such as the question of whether or not it is safe to remove a particular barrier statement from an MPI program, will also receive attention. The limitations of testing and other dynamic methods will also be explored.

The basic tasks involved in model checking will then be introduced: the construction of a *model* of the program being verified, the formulation of one or more *properties* of the model, and the use of automated algorithmic techniques for checking that every execution of the model satisfies the property. The limitations of model checking will also be discussed; these include the *state explosion problem* and the problem of accurately constructing appropriate models of programs. It will be emphasized that, while modeling requires a certain degree of

* This material is based upon work supported by the National Science Foundation under Grant No. 0541035.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 13–14, 2007.

skill, it is not more difficult than programming and with a little practice most MPI programmers can become very effective modelers.

The remainder of this part of the tutorial will consist of a tool demonstration, which will also introduce the main example used throughout the tutorial, the **diffusion** program.

2. Language Basics. Some knowledge of programming languages carries over into modeling languages, but modeling differs in several significant ways. In this part of the tutorial, the basic syntax and semantics of the MPI-Spin input language will be described in a progressive, methodical way. The description will start with those syntactic elements dealing with process declaration and management, then move on to variables and types, then expressions, and finally the different types of statements provided by the language. Throughout, the **diffusion** example will be used to illustrate the various language constructs.

3. Using MPI-Spin. This part will begin with a discussion of *abstraction* and how the choice of appropriate abstractions can lead to models that are both *efficient* and *conservative*. These notions will be defined precisely and illustrated using **diffusion** and **matmat**, a program that computes the product of two matrices using a master-slave pattern. In the latter example, the consequences of using various abstractions for the variables in the program will be explored. It will be shown that different choices are appropriate for verifying different properties of **matmat**.

Once a model has been constructed, the MPI-Spin tool itself must be executed on that model. As is the case for any complex tool (such as a compiler), there are many options, parameters, and flags available to the user. The most commonly used options will be described and their effects demonstrated using the examples introduced previously.

This will be followed by a "hands-on" exercise in which MPI-Spin is used to explore the consequences of modifications to the **diffusion** code.

4. Verifying Correctness of Numerical Computations. The fourth part of the tutorial deals with a recent and exciting development in the field of model checking for HPC: techniques using symbolic execution to verify properties of the numerical computations carried out by parallel programs. These techniques are supported in MPI-Spin through an abstract datatype `MPI_Symbolic` together with a number of operations on that type, such as `SYM_add` and `SYM_multiply`. The idea is to model the inputs to the program as symbolic constants x_i and the output as a vector of symbolic expressions in the x_i. The output vector can be analyzed for a number of purposes. Most importantly, it can be compared against the symbolic output vector produced by a trusted sequential version of the program. In this way, model checking can be used to show that for *all inputs*, and for *all possible executions* of the parallel program on the input, the parallel program will produce the same results as the sequential one. This technique will be illustrated using **matmat** and, if time permits, a more complex example implementing the Gaussian elimination algorithm.

Advanced MPI Programming

Julien Langou[1] and George Bosilca[2]

[1] Department of Mathematical Sciences,
University of Colorado, Denver, CO, USA
langou@math.cudenver.edu
[2] Innovative Computing Laboratory,
University of Tennessee, Knoxville, TN, USA
bosilca@cs.utk.edu

MPI provides a large range of features allowing various approaches for parallel computing. This tutorial will present interesting features from the MPI-1 standard. These features extend the user knowledge about MPI way beyond the few basic standard functions, giving them the opportunity to implement better, simpler and potentially faster parallel algorithms. This tutorial will cover several features from medium level to advanced of the MPI-1 standard to enable users to exploit fully MPI.

Identifying the bottlenecks. Once an initial version of a parallel application exists, the first question raised is how to improve it. While a large improvement might come from algorithmic modifications, there is usually a large space of improvements that might come from a correct useage of the MPI specification. Before starting any optimization, the user has first to identify the portions of the application that have the greatest potential for improvement. The basic approach is to use the MPI profiling layer. Unfortunately, this layer lacks the transparency required to show the effective data transfer over the network, missing a rich set of information about the application behavior. PERUSE is a more suitable layer to get access to all the hidden information available from the MPI libraries.

Optimizing memory accesses. The MPI specification introduces a number of functions for creating MPI data-types. These data-types represent memory layouts and can be used for message exchange; both in point-to-point or collective communications. Choosing the right data-type is a performance critical choice as its impact on the communication performance is tremendous.

Advanced point-to-point communications. While blocking communications are easier to masterize, they introduce one synchronization stage per communication which might be harmful to the overall performance of the application. Moving away from this blocking model, by using non-blocking point-to-point communications, usually allows the user for a better overlap between multiple communications leading to a shorter execution time of the parallel application.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 15–16, 2007.

Collective communications. The collective communications provides a highly optimized variety of algorithms for global data exchange between a set of MPI processes. These algorithms cover some of the most generic collective communications such as broadcast, barrier, etc. Some of them (i.e. those derived from the reduce operation) can be extended using user-provided operations.

For each item in the above list, we will provide attendees with small and comprehensible example codes that have been identified as critical part of larger relevant application codes. Starting from these small codes, we will show how to exploit MPI functionallities in order to improve simplicity and efficiency of the application. The example codes and their modifications will be made available to the attendees.

Full Bandwidth Broadcast, Reduction and Scan with Only Two Trees

Peter Sanders[1], Jochen Speck[1], and Jesper Larsson Träff[2]

[1] Universität Karlsruhe
Am Fasanengarten 5, D-76131 Karlsruhe, Germany
sanders@ira.uka.de
[2] NEC Laboratories Europe, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
traff@ccrl-nece.de

Abstract. We present a new, simple algorithmic idea for exploiting the capability for bidirectional communication present in many modern interconnects for the collective MPI operations broadcast, reduction and scan. Our algorithms achieve up to *twice* the bandwidth of most previous and commonly used algorithms. In particular, our algorithms for reduction and scan are the currently best known. Experiments on clusters with Myrinet and InfiniBand interconnects show significant reductions in running time for broadcast and reduction, for reduction even close to the best possible factor of two.

1 Introduction

The *Message Passing Interface* (MPI) [12] offers a set of *collective communication and computation operations* that are eminently useful for expressing parallel computations. It is therefore important that MPI libraries implement these operations as efficiently as possible for the intended target architectures. Hence, algorithms that can fully exploit the given communication capabilities are needed. Consequently, recent years has seen a lot of activity on algorithms and implementations for MPI collectives for different communication architectures, see [3,8,10,13] to mention but a few.

In this paper we give new algorithms with implementations for three important MPI collectives, namely `MPI_Bcast` (broadcast), `MPI_Reduce` (reduction to root) and `MPI_Scan`/`MPI_Exscan` (parallel prefix). The new algorithms are able to fully exploit bidirectional communication capabilities as offered by modern interconnects like InfiniBand, Myrinet, Quadrics, and the NEC IXS, and thus in contrast to many commonly used algorithms for these operations have the potential of achieving the full bandwidth offered by such interconnects. For broadcast this has previously been achieved also by other algorithms [1,6,16], but these are typically more complicated and do not extend to the reduction and parallel prefix operations. We believe the results achieved for reduction and parallel prefix to be the theoretically currently best known. The algorithms are still simple to implement, and have been implemented within the framework of NEC proprietary MPI libraries.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 17–26, 2007.

2 Two Pipelined Binary Trees Instead of One

To explain the new algorithm we focus on the `MPI_Bcast` operation, and illustrate the improvement achieved by comparing to a *linear pipeline* and a *pipelined binary tree*. We let p denote the number of processors which are numbered from 0 to $p-1$, and m the size of the data to be broadcast from a given root processor r. As in `MPI_Bcast` we assume that all processors know p, r and m. We assume that communication is homogeneous, one-ported, and bidirectional in the sense that each processor can at the same time send a message to a processor and receive a message from another, possibly different processor.

Assume first that $p = 2^h - 1$ for some tree height $h > 1$. A binary tree broadcast algorithm uses a balanced, ordered binary tree rooted at processor r. To broadcast, the root sends its data to its left and right child processors, and upon receiving data each processor which is not a leaf sends data to its left and right child processors in that order. As can be seen the rightmost leaf receives data at communication step $2h$. Assuming that the time to transfer data of size m is $\alpha + \beta m$, the total broadcast time is $2h(\alpha + \beta m)$. The binary tree algorithm can be pipelined, and sending instead the m data as N blocks of size m/N yields a broadcast time (for the rightmost leaf) of $2(h+N-1)(\alpha+\beta m/N)$ (even exploiting bidirectional communication: each interior processor receives a new block from its parent at the same time as it sends a block to its right child). Balancing the terms $N\alpha$ and $(h-1)\beta m/N$ yields a best broadcast time of $2(h-1)\alpha+4\sqrt{(h-1)\alpha}\sqrt{\beta m}+\mathbf{2}\beta\mathbf{m}$. This is roughly a factor of two from the lower bound of $\min\{\alpha h, \beta\mathbf{m}\}$.

Using instead a linear pipeline in which processor i receives a new block from processor $i-1$ and a the same time sends the previous blocks on to processor $i+1$, a broadcast time (for the last processor) of $(p-2)\alpha+2\sqrt{(h-1)\alpha}\sqrt{\beta m}+\beta\mathbf{m}$ can be achieved. This algorithm has asymptotically optimal broadcast time βm but at the cost of a large latency term $(p-2)\alpha$ and is therefore interesting only when m is large compared to p.

We are interested in algorithms that combine the low latency of the pipelined binary tree and the optimal time achieved by the linear pipeline, while retaining as far as possible the implementation simplicity of these algorithms (as well as other properties such as low demands on bisection bandwidth, and embeddability in non-homogeneous networks).

The problem with the pipelined binary tree algorithm is that the bidirectional communication is only exploited in every second communication step for the interior node processors and not at all for the root and leaf processors. In particular, the leaves are only receiving blocks of data. We propose to use two binary trees simultaneously in order to achieve a more balanced use of the communication links. The two trees are constructed in such a way that the interior nodes of one tree correspond to leaf nodes of the other. This allows us to take full advantage of the bidirectional communication capabilities. In each communication step a processor receives a block from its parent in one of the two trees and sends the previous block to one of its children in the tree in which it is an interior node. To make this work efficiently, the task is to devise a construction

of the two trees together with a schedule which determines for each time step[1] from which parent a block is received and to which child the previous block is sent so that each communication step consists in at most one send and at most one receive operation for each processor.

For now we (w.l.o.g) assume that $r = p-1$ and construct the two binary trees T_1 and T_2 over processors $0 \ldots p-2$ as follows. Let 2^h be the smallest power of two larger or equal to $p-1$ and let $P = 2^h - 1$.

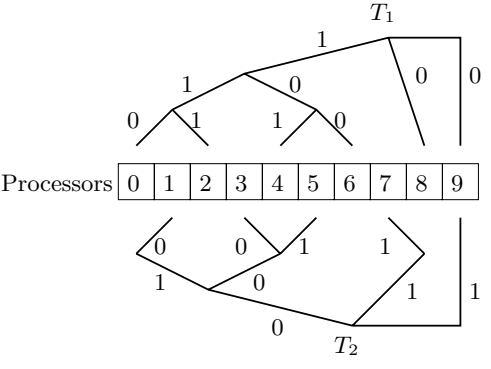


Fig. 1. The two inorder binary trees T_1 (top to bottom) and T_2 (bottom to top) with broadcast root $r = 9$ for $p = 10$. A 0/1 coloring is also shown.

1. Construct an inorder numbered balanced tree of size P, such that that the leaves have even numbers $0, 2, \ldots P-2$.
2. If $P > p-1$ eliminate nodes starting from the highest numbered node u as follows: if u is a leaf simply remove it; otherwise set the parent of u's (only) child to be the parent of u and remove u. This tree is T_1.
3. Construct an inorder numbered balanced tree of size P.
4. Shift the tree one position to the left, e.g. rename node u to $u-1$. Remove the leftmost leaf -1. In this tree leaves have odd numbers $1, 3, \ldots$.
5. If $P-1 > p-1$ eliminate nodes from the right as in Step 2. This tree is T_2.

To complete the construction for the broadcast algorithm, the roots of the two trees are connected to the broadcast root node $p-1$. An example of the construction for $p = 10$ is shown in Figure 1. The construction obviously has the desired property that leaves of T_1 are interior nodes of T_2 and vice versa.

To use the two trees for broadcast, the idea is to pipeline half the data through T_1, and the other half of the data through T_2. With bidirectional communication capabilities, the two pipelines can run concurrently, yielding a broadcast time of $2[2(h-1)\alpha + 4\sqrt{(h-1)\alpha}\sqrt{\beta m/2} + 2\beta m/2] = 4(h-1)\alpha + 4\sqrt{(h-1)\alpha}\sqrt{\beta m/2} + \beta \mathbf{m}$. This is half the time of the pipelined binary tree at the expense of only twice the (still) logarithmic latency.

[1] Note that no explicit global synchronization is necessary. Point-to-point communication suffices to implicitly synchronize the processors to the extent necessary.

The left to right order of the children is not sufficient to avoid that a node in the same communication step receives a block from both of its parents (which would compromise the analysis and slow down the algorithm). We need an even-odd (0/1) coloring of the parent to child edges without such conflicts. Even (0) edges are then used in even communication steps, odd (1) edges in odd steps. More precisely we need the following Lemma.

Lemma 1. *The edges of T_1 and T_2 can be colored with colors 0 and 1 such that*

1. *no PE is connected to its parent nodes in T_1 and T_2 using edges of the same color and*
2. *no PE is connected to its children nodes in T_1 or T_2 using edges of the same color.*

Proof. Consider the bipartite graph $B = (\{s_0, \ldots, s_{p-1}\} \cup \{r_1, \ldots, r_{p-1}\}, E)$ where $\{s_i, r_j\} \in E$ iff j is a successor of i in T_1 or T_2. This graph models the sender role of processor i with node s_i and the receiver role of processor i with node r_i. By the construction of T_1 and T_2, B has maximum degree two, i.e., B is a collection of paths and *even* cycles. Hence, the edges of B can be two-colored by just traversing these paths and cycles.

Computing the coloring as described in the proof can be done in $O(p)$ steps. In the appendix we outline how (with another construction of the two trees) each processor i can compute all relevant information (neighbors in the trees, colors of incident edges) in time $O(\log p)$ given only p and i.

2.1 Broadcast

If the broadcast root is not $r = p - 1$ as in the above, we simply renumber each processor i to instead play the role of processor $(i - (r + 1)) \bmod p$. To be useful in MPI libraries the coloring should be done at communicator construction time so that the coloring time is amortized over all subsequent broadcast operations on the communicator.

For broadcast the idea of using two trees to improve bandwidth was previously introduced in [5], but the need for coloring was not realized (due to the TCP/IP setting of this work).

2.2 Reduction

Let $\oplus$ denote an associative (possibly commutative) binary operation, and let processor i have a data vector m_i of size m. The reduction to root operation `MPI_Reduce` computes $\oplus_{i=0}^{p-1} m_i$ and stores the result at the root processor r. Assuming $r = 0$ or $r = p-1$, the two tree construction can be used for reduction to root by reversing the flow of data from the leaves towards the root in both of the trees. Since the trees are inorder numbered the reduction order can be maintained such that only associativity of $\oplus$ is exploited. We have the following Theorem.

Theorem 1. *For $r = 0$ or $r = p - 1$ reduction to root of data m_i (each of size m) can be done in communication time $4(h-1)\alpha + 4\sqrt{(h-1)\alpha}\sqrt{\beta m/2} + \beta m$. The amount of data reduced per processor is $2m$.*

If the operator $\oplus$ is commutative the result can be achieved for any root r.

2.3 Scan

The (inclusive) parallel prefix operation `MPI_Scan` computes $\oplus_{i=0}^{j} m_i$ for each processor j. Using an inorder binary tree parallel prefixes can be computed by first performing an *up-phase* in which each interior node computes a partial sum $\oplus_{i=\ell}^{r}$ for left- and rightmost leaves ℓ and r, followed by a *down-phase* in which prefixes of the form $\oplus_{i=0}^{\ell-1}$ are sent down the tree and allow each processor to finish computing its parallel prefix. Both phases can be pipelined, and for each N blocks can be completed in $2h - 1 + 2(N-1)$ communication steps. This is explained in more detail in [7,11]. With the two tree construction two up-phases and two down-phases can take place simultaneously. This halves the number of steps for N blocks for each phase. The total number of steps required for the parallel prefix computation is therefore $4h - 2 + 2(N-2)$. This is roughly two thirds the $4h - 2 + 3(N-1)$ steps required by the *doubly pipelined parallel prefix algorithm* described in [11].

Theorem 2. *The parallel prefix operation on data m_i can be done in communication time $2((2h-3)+2\sqrt{(2h-3)\alpha}\sqrt{\beta m/2}+\beta m$. The amount of data reduced per processor is (at most) $3m$.*

3 Experimental Results

The two tree algorithms for `MPI_Bcast` and `MPI_Reduce` have been implemented within proprietary NEC MPI implementations. Experiments comparing the bandwidth achieved with the new algorithms to other, commonly used broadcast and reduction algorithms have been conducted on a small AMD Athlon based cluster with Myrinet 2000 interconnect, and a larger Intel Xeon based InfiniBand cluster. Bandwidth is computed as data size m divided by the time to complete for the slowest process. Completion time is the smallest measured time (for the slowest process) over a small number of repetitions. We give only results for the case with one MPI process per node, thus the number of processors p equals the number of nodes of the cluster.

3.1 Broadcast

We compare to the following algorithms:

- *Circulant graph*first presented in [16]. This algorithm has asymptotically optimal completion time, and only half the latency of the two tree algorithm presented here, but is significantly more complex and requires a more involved precomputation than the simple coloring needed for the two tree algorithm.

– *Scatter-allgather* for broadcast [2] as developed in [14]. We also contrast to the implementation of this algorithm in MVAPICH.
– Simple *binomial tree* as in the original MPICH implementation [4].
– Pipelined binary tree
– Linear pipeline

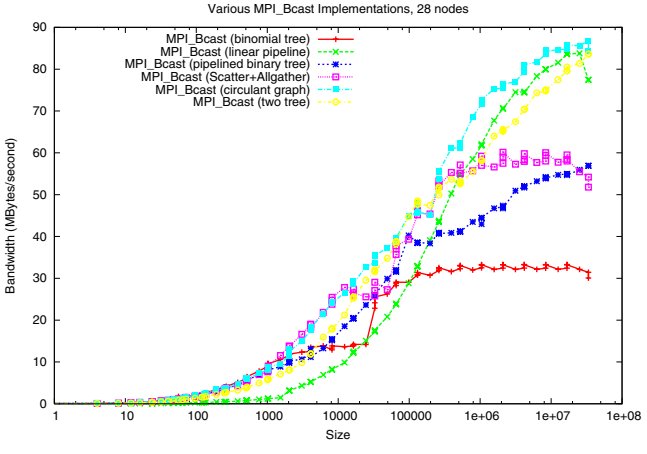


Fig. 2. Broadcast algorithms on the AMD/Myrinet cluster, 28 nodes

Bandwidth results for the two systems are shown in Figure 2 and Figure 3. On both systems the two tree algorithm asymptotically achieves the same bandwidth as the optimal circulant graph algorithm, but can of course not compete for small problems where the circulant graph algorithm degenerates into a binomial tree which has only half the latency of the binary tree. Even for large m (up to 16MBytes) both algorithms fare better than the linear pipeline, although none of the algorithms have reached their full bandwidth on the InfiniBand cluster. On the Myrinet cluster the algorithms achieve more than 1.5 times the bandwidth of the scatter-allgather and pipelined binary tree algorithms. For the Myrinet cluster where we also compared to the simple binomial tree a factor 3 higher bandwidth is achieved for 28 processors.

The two tree broadcast algorithm is a serious candidate for improving the broadcast bandwidth for large problems on bidirectional networks. It is arguably simpler to implement than the optimal circulant graph algorithm [16], but have to be combined with a binomial tree algorithm for small to medium sized problems. Being a pipelined algorithm with small blocks of size $\Theta(\sqrt{m})$ it is also well suited to implementation on SMP clusters [15].

3.2 Reduction

We compare to the following algorithms:

– Improved *butterfly* [9]

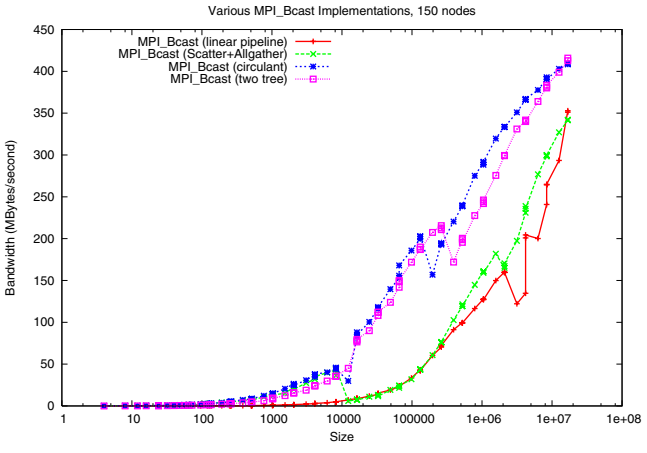


Fig. 3. Broadcast algorithms on the Xeon/InfiniBand cluster, 150 nodes

- *Binomial tree*
- *Pipelined binary tree*
- *Linear pipeline*

Bandwidth results for the two systems are shown in Figure 4 and Figure 5. The two tree algorithm achieves about a factor 1.5 higher bandwidth than the second best algorithm which is either the pipelined binary tree (on the Myrinet cluster) or the butterfly (on the InfiniBand cluster). On both systems the linear pipeline achieves an even higher bandwidth, though, but problem sizes have to be larger than 1MByte (for $p = 28$ on the Myrinet cluster), or 16Mbyte (for $p = 150$ on the InfiniBand cluster), respectively. For smaller problems the linear pipeline is inferior and should not be used. On the InfiniBand cluster there is a considerable difference of almost a factor 2 between the two implementations of the butterfly algorithm (with the implementation of [9] being the faster). The sudden drop in bandwidth for the butterfly algorithm on the Myrinet cluster is due to a protocol change in the underlying point-to-point communication, but for this algorithm it is difficult to avoid getting into the less suitable protocol domain. The pipelined algorithms give full flexibility in the choice of block sizes and such effects can thus better be countered.

3.3 Coloring

Table 1 compares the preprocessing times for a simple linear time implementation of two tree coloring and the $O(p \log p)$ time algorithm for computing the block schedule required for the circulant graph algorithm from [16]. The coloring algorithm is always faster than the block scheduling algorithm. It is to be expected that with the logarithmic time coloring algorithm, the speed difference would become quite dramatic for large p. However, for currently used machine sizes, both scheduling times are not a big issue when they are only needed once for each communicator.

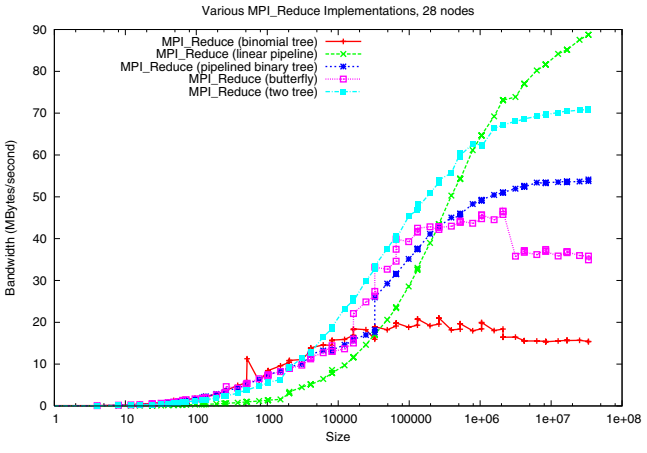


Fig. 4. Reduction algorithms on the AMD/Myrinet cluster, 28 nodes

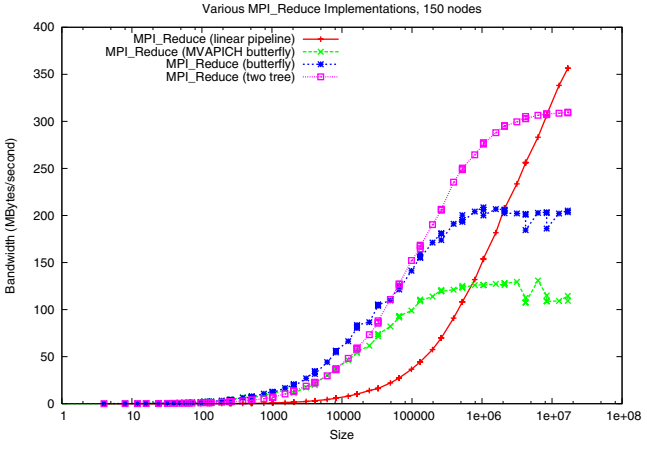


Fig. 5. Reduction algorithms on the Xeon/InfiniBand cluster, 150 nodes

Table 1. Computing times in microseconds on an AMD 2.1GHz Athlon processor for the precomputation of double-tree coloring and block schedule

Processors	Two tree 0/1-coloring	Circulant graph block schedule
100	71.75	99.15
1000	443.37	1399.43
10000	13651.42	20042.28
100000	209919.33	248803.58
1000000	1990228.74	3074909.75

4 Conclusion

We presented a new, simple algorithmic idea for broadcast, reduction and parallel prefix operations as found in MPI. The theoretical result and achieved performance for `MPI_Bcast` is similar to that achieved by other, recent, but more complicated algorithms [16]. The theoretical results for reduction to root and parallel prefix are presumably the best currently known, and the implementation of `MPI_Reduce` show a significant improvement over several other algorithms. We expect a similar result for the `MPI_Scan` implementation.

References

1. Bar-Noy, A., Kipnis, S., Schieber, B.: Optimal multiple message broadcasting in telephone-like communication systems. Discrete Applied Mathematics 100(1–2), 1–15 (2000)
2. Barnett, M., Gupta, S., Payne, D.G., Schuler, L., van de Geijn, R., Watts, J.: Building a high-performance collective communication library. In: Supercomputing'94, pp. 107–116 (1994)
3. Chan, E.W., Heimlich, M.F., Purkayastha, A., van de Geijn, R.A.: On optimizing collective communication. In: IEEE International Conference on Cluster Computing CLUSTER 2004, IEEE Computer Society Press, Los Alamitos (2004)
4. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22(6), 789–828 (1996)
5. Happe, H.H., Vinter, B.: Improving TCP/IP multicasting with message segmentation. In: Communicating Process Architectures (CPA 2005) (2005)
6. Kwon, O.-H., Chwa, K.-Y.: Multiple message broadcasting in communication networks. Networks 26, 253–261 (1995)
7. Mayr, E.W., Plaxton, C.G.: Pipelined parallel prefix computations, and sorting on a pipelined hypercube. Journal of Parallel and Distributed Computing17, 374–380 (1993)
8. Pjesivac-Grbovic, J., Angskun, T., Bosilca, G., Fagg, G.E., Gabriel, E., Dongarra, J.: Performance analysis of MPI collective operations. In: International Parallel and Distributed Processing Symposium (IPDPS 2005), Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO) (2005)
9. Rabenseifner, R., Träff, J.L.: More efficient reduction algorithms for message-passing parallel systems. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3241, pp. 36–46. Springer, Heidelberg (2004)
10. Ritzdorf, H., Träff, J.L.: Collective operations in NEC's high-performance MPI libraries. In: International Parallel and Distributed Processing Symposium (IPDPS 2006), p. 100 (2006)
11. Sanders, P., Träff, J.L.: Parallel prefix (scan) algorithms for MPI. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 49–57. Springer, Heidelberg (2006)
12. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI – The Complete Reference, 2nd edn. The MPI Core, vol. 1. MIT Press, Cambridge (1998)

13. Thakur, R., Gropp, W.D., Rabenseifner, R.: Improving the performance of collective operations in MPICH. International Journal on High Performance Computing Applications 19, 49–66 (2004)
14. Träff, J.L.: A simple work-optimal broadcast algorithm for message-passing parallel systems. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3241, pp. 173–180. Springer, Heidelberg (2004)
15. Träff, J.L., Ripke, A.: An optimal broadcast algorithm adapted to SMP-clusters. In: Di Martino, B., Kranzlmüller, D., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3666, pp. 48–56. Springer, Heidelberg (2005)
16. Träff, J.L., Ripke, A.: Optimal broadcast for fully connected networks. In: Yang, L.T., Rana, O.F., Di Martino, B., Dongarra, J.J. (eds.) HPCC 2005. LNCS, vol. 3726, pp. 45–56. Springer, Heidelberg (2005)

Logarithmic Time Scheduling Algorithm

The algorithm below assumes that p is even and T_2 is the mirror image of T_1. The tree construction is a straight forward recursive algorithm. In the full paper we prove the (non obvious) correctness of the simple coloring algorithm below.

Function $\mathsf{inEdgeColor}(p, i, h)$
  If i is the root of T_1 **Then Return** 1
  While $i \,\mathsf{bitand}\, 2^h = 0$ **Do** h++ $\quad$ −− compute height
  $i' := \begin{cases} i - 2^h \text{ if } 2^{h+1} \,\mathsf{bitand}\, i = 1 \vee i + 2^h > p \\ i + 2^h \text{ otherwise} \end{cases}$ $\quad$ −− compute parent of i
  Return $\mathsf{inEdgeColor}(p, i', h) \,\mathsf{xor}\, (p/2 \bmod 2) \,\mathsf{xor}\, [i' > i]$

Process Cooperation in Multiple Message Broadcast

Bin Jia

IBM Advanced Clustering Technology Team
Poughkeepsie, NY 12601
binj@us.ibm.com

Abstract. We present a process cooperation algorithm for broadcasting m messages among n processes, $m \geq 1, n \geq 1$, in one-port fully-connected communication systems. In this algorithm, the n processes are organized into $2^{\lfloor \log n \rfloor}$ one- or two-process units. Messages are broadcast among the units according to a basic communication schedule. Processes in each two-process unit cooperate to carry out the basic schedule in a way that at any step, either process has at most one message that the other has not received. This algorithm completes the broadcast in $\lceil \log n \rceil + m - 1$ communication steps, which is theoretically optimal. Empirical study shows that it outperforms other widely used algorithms significantly when the data to broadcast is large. Efficient communication schedule construction is a salient feature of this algorithm. Both the basic schedule and the cooperation schedule are constructed in $O(\log n)$ bitwise operations on process ranking.

Keywords: Broadcast, process cooperation, MPI collective communication, one-port fully-connected system, communication schedule.

1 Introduction

Broadcast is one of the most important communication primitives in parallel and distributed systems. The MPI standard [11] defines a MPI_BCAST interface for single source broadcast operation, in which data available at the source process is copied to all other processes. When the amount of data to broadcast is large, the data is often split into multiple messages, which are pipelined individually for better bandwidth utilization [2,12,1,3,4,13,9].

In this paper, we investigate the problem of broadcasting m messages among n processes in one-port fully-connected communication systems, $m \geq 1, n \geq 1$. Distances between any two processes in such a system are equal. Every process can send a message to one process and receive a message from another process in a single communication step. It takes the source process at minimum $m-1$ steps before the last message can be sent and at least $\lceil \log n \rceil$ steps are needed for the last message to reach all processes. Therefore the lower bound on communication steps is $\lceil \log n \rceil + m - 1$. A multiple message broadcast algorithm is optimal if the number of steps it needs to complete the broadcast matches the lower bound.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 27–35, 2007.

The communication schedule of a multiple message broadcast algorithm defines which part of the data should a message contain; when to send the message; and the source/destination processes of the message. Widely adopted algorithms such as the *pipelined chain* algorithm and the *pipelined binary tree* algorithm have simple communication schedules that can be easily determined on-line. Those algorithms, however, are not optimal in the one-port fully-connected system. Other algorithms [1,12] exploit complicated communication schedules to achieve optimal broadcast time. Except for in special cases such as with power-of-two-process [6,12], the schedule construction in those algorithms is so expensive that it can only be done off-line, which hinders their incorporation in communication libraries.

We present an optimal process cooperation algorithm with efficient communication schedule construction. The algorithm includes a *basic schedule* for broadcasting among power-of-two processes and a *cooperation schedule*. The n processes are organized into n' one-process or two-process units, where $n' = 2^{\lfloor \log n \rfloor}$. Following the cooperation schedule, processes in each two-process unit cooperate to carry out the basic schedule. At any step, either process has at most one message that the other process has not received. This schedule can be constructed efficiently on-line in only $O(\log n)$ bit operations on process ranking. We have implemented and incorporated this algorithm in IBM Parallel Environment/MPI [5].

The rest of the paper is organized as follows. Section 2 overviews existing multiple message broadcast algorithms. Section 3 describes the construction of the basic schedule for power-of-two processes. Section 4 details the process cooperation in multiple message broadcast and its schedule construction. Correctness proof is also given. Performance results of MPI_BCAST implementation using the process cooperation algorithm are reported and compared with other commonly used algorithms in Section 5. Section 6 concludes the paper and discusses future study directions.

2 Related Work

A straightforward pipeline approach for multiple message broadcast is to arrange the processes into a communication chain with the source at the head and pipeline the messages along the chain. $m + n - 2$ steps are needed by this type of pipelined chain algorithm to complete the broadcast since it takes a message $n - 1$ steps to reach the end of the chain. The pipelined binary tree algorithm reduces the n dependent cost to $2 \times \lceil \log n \rceil$ steps by pipelining messages along a binary tree of processes. However, it takes twice as many steps before the source can send the last message. Total number of steps required is therefore $2 \times (m + \lceil \log n \rceil - 1)$. Further tradeoff between the n dependent steps and the m dependent steps can be made in the *fractional tree* algorithm [10] — a generalization of the pipelined chain and binary tree algorithms where node in the binary tree is replaced by a chain of processes. In the *scatter-allgather* algorithm [2], data is split into $m = n$ messages. The messages are first scattered

among the processes and then each process collects the rest of the messages from other processes through an allgather operation. Therefore it takes the scatter-allgather algorithm $2 \times (n-1)$ steps to complete the broadcast. These algorithms are easy to implement but not optimal.

Johnsson and Ho [6] have developed an *Edge-disjoint Spanning Binomial Tree* (ESBT) algorithm for hypercube systems, which can be easily embedded into fully-connected system. The processes are organized into $\log n$ binomial trees each consisting of all n processes. The source sends messages to different trees in a round robin fashion. Messages are passed along the trees in parallel and free of edge contention. The algorithm is optimal but it only works when n is a power-of-two.

Bar-Noy et al. [1] have sketched an optimal multiple message broadcast algorithm for any number of processes as a by-product of their study on the problem in fully-connected systems using the *telephone* model. The algorithm is complicated when n is not a power-of-two since it is based on the more restrictive telephone model. The focus of both their algorithm and the ESBT algorithm is more on showing the existence of an optimal schedule but less attention is paid to the practical implementation issue of how to construct the schedule.

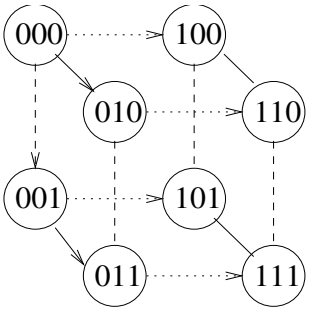
Träff and Ripke [12] have recently extended the ESBT algorithm to any number of processes by allowing incomplete binomial trees and implemented the algorithm in a MPI library. They have developed an $O(n \times \log n)$ greedy algorithm to construct the optimal but complex schedule. The construction is so expensive that it can only be done off-line at MPI communicator creation and cached with the MPI communicator object for use in MPI_BCAST operation.

3 Efficient Schedule Construction for Power of Two Processes

In this section, we describe an optimal algorithm for broadcasting m messages among $n = 2^q$ processes. It will be used as the base for our process cooperation algorithm. In the rest of the paper, the m messages are denoted by $M_0, M_1, \ldots$ through M_{m-1} and the n processes are ranked from 0 through $n-1$. Without loss of generality, let process 0 be the source process of the broadcast.

The algorithm begins with a *setup* phase consisting of q steps. Processes are organized into a binomial tree rooted at process 0. Process 0 sends a distinct message to one of its children during each step. Each of the other processes forwards the message received from its parent to all its children. A *pipeline* phase of $m-1$ steps then follows. During each step, every process exchanges messages with a partner process. Figure 1 (right) outlines the algorithm. This algorithm works similarly to other known optimal algorithms [1,6,12]. Contribution of this paper on this part is an efficient construction of the communication schedule, referred to hereafter as the basic schedule.

To construct the basic schedule, r, s, t, $Parent$, $Child$ and $Partner$ in Figure 1 (right) need to be determined. Simple bit operations on process ranking are performed for this purpose. Let $(i_{q-1} i_{q-2} \ldots i_0)$ be the binary representation of



if $i = 0$ **then**
  for $j \leftarrow 0,\ q-1$ **do**
    send M_j to $Child(0, j)$
otherwise
  receive M_r from $Parent(i)$
  for $j \leftarrow 0,\ q-l-2$ **do**
    send M_r to $Child(i, j)$
for $j \leftarrow 0,\ m-2$ **do**
  receive M_t from $Partner(i, j)$;
  send M_s to $Partner(i, j)$

Fig. 1. Left: example of process relationship, $n = 8$. Arrows point to children processes. Dashed, solid and dotted lines denote partnership in pipeline step 0, 1, and 2 respectively. Right: algorithm outline.

process i. r and l can be determined easily by scanning the binary representation to locate the rightmost nonzero bit: i_r and the leftmost nonzero bit: i_l. Also define $l = -1$ for $i = 0$, parent and the j-th child of process i are given by:

$$Parent(i) = (i_{q-1} \dots \overline{i_l} \dots i_0) \ ,$$
$$Child(i, j) = (i_{q-1} \dots \overline{i_{l+j+1}} \dots i_0) \ .$$

where $0 \leq j \leq q - l - 2$.

Let $e = (j \mod q)$, partner process of process i during step j, $0 \leq j \leq m-2$, of the pipeline phase is given by: ,

$$Partner(i, j) = (i_{q-1} \dots \overline{i_e} \dots i_0) \ .$$

Figure 1 (left) shows an example of the parent-children relationship in the setup phase and the partnership in the pipeline phase.

For process 0, $s = j + q$ and $t = j$. In fact, process 0's receive and its partner's send can be replaced by no-op. To determine s and t for process i that is neither the source nor its partner, we first build a q-element array Dis to store the distance between each bit and the next nonzero bit to the left by scanning the binary representation:

- let $f = 1$, $g = 1$.
- while $g \leq q - 1$,
  - if $i_g = 1$, fill up a section of the array: $Dis[h] = g - h$ for $f - 1 \leq h < g$ and then let $f = g + 1$.
  - let $g = g + 1$
- if $i_{q-1} = 1$, let $Dis[q-1] = r + 1$. Otherwise, let $Dis[h] = r + q - h$ for $f - 1 \leq h \leq q - 1$.

Table 1 gives an example of the Dis array. Finally, at pipeline step j, if $i_e = 0$, then $s = j + Dis[e]$, $t = j$; otherwise it is the other way around.

It can be seen that in the pipeline phase, the communication pattern repeats every q steps and all processes have M_j after step j. When the schedule requires a message labeled beyond $m - 1$ to be sent/received, message M_{m-1} is

sent/received instead. The source injects M_{m-1} during pipeline step $m-1-q$ and keeps doing the same thereafter. Therefore the broadcast completes in $\log n + m - 1$ steps, matching the lower bound. Correctness of this basic schedule can be proved by induction. Both the time and space complexity of building the schedule are $O(\log n)$.

4 Process Cooperation

In this section, we extend the algorithm described in Section 3 to arbitrary number of processes. The ides is to organize the n processes into n' units each having one or two processes, $q = \lfloor \log n \rfloor$ and $n' = 2^q$. The basic schedule is applied to broadcast among the n' units. Processes in each two-process unit cooperate in: 1) sending/receiving messages to/from other units according to the basic schedule; and 2) passing message between themselves such that at any step, either process has at most one message that the other process has not received. For this purpose, we construct a process cooperation schedule as an extension to the basic schedule.

First we define a simple scheme to organize processes into units. For process i, define:

$$Co(i) = \begin{cases} i - n' + 1 & i \geq n' \\ n' - 1 + i & 0 < i \leq n - n' \\ i & otherwise \end{cases}$$

$$Rep(i) = \begin{cases} i & i < n' \\ Co(i) & otherwise \end{cases}$$

Also, the unit to which i belongs is defined as $Unit(i)$. The first row of Table 2 shows an example of the scheme.

Process i participates in the setup phase if and only if $i < n'$. The setup phase is the same as in the basic schedule. During each step of the pipeline phase, on the other hand, $Rep(i)$'s binary representation is used in determining s, t, and $Partner(i, j)$, instead of i's. Since $Rep(i) < n'$, q-bit binary representation is still sufficient.

In a two-process unit, one process is called the *output* of the unit and the other the *input*. The output sends a message to the partner unit and the input receives a message from the partner unit. The input also passes a message it received during the setup phase or previous steps of the pipeline phase to the output. In a one-process unit, the process is both the input and the output. Process's role changes from step to step and its determination is the core of the cooperation schedule.

Initial roles are assigned at the beginning of the pipeline phase: $Rep(i)$ is the output of $Unit(i)$ and $Co(Rep(i))$ is the input. If bit $(j \mod q)$ in $Rep(i)$'s binary representation is equal to 1, the input and output of $Unit(i)$ switch roles after pipeline step j. Let $Switch(i, j)$ be the number of nonzero bits in $Rep(i)$

Table 1. Example of Dis, and $Switch$, $i = 406, n = 512$

j	8	7	6	5	4	3	2	1	0
i_j	1	1	0	0	1	0	1	1	0
$Dis[j]$	2	1	1	2	3	1	2	1	1
$Switch(i, j)$	5	4	3	3	3	2	2	1	0

between bit 0 and bit ($j \mod q$), then according to the rule of role switch, the number of role switches for process i before pipeline step j is:

$$u = Switch(i, q-1) \times \lfloor (j-1)/q \rfloor + Switch(i, (j-1)) \ .$$

Process i's role during step j is the same as its initial assignment if u is even. It is the opposite if u is odd. Table 1 shows an example of the $Switch$ function.

The input and output of the partner unit also need to be determined if the partner unit is a two-process unit. Since there is only one bit difference between the binary representations of $Rep(i)$ and $Partner(i, j)$, $Partner(i, j)$'s role during pipeline step j is the same as its initial assignment if $v = u + \lfloor (j-1)/q \rfloor$ is even. It is the opposite otherwise.

It can be seen that both the time and space complexity of the role determination are $O(\log n)$. With the roles determined, the pipeline phase of the process cooperation algorithm can be described as follows:

for $j \leftarrow 0, m-2$ **do**
  output of $Unit(i)$ sends M_s to input of $Unit(Partner(i, j))$
  input of $Unit(i)$ receives M_t from output of $Unit(Partner(i, j))$
  if $j > 0$ and $i \neq Co(i)$ **then**
    input of $Unit(i)$ passes M_{j-1} to the output of $Unit(i)$
if $i \neq Co(i)$ **then**
  input of $Unit(i)$ passes M_{m-2} to the output of $Unit(i)$
  output of $Unit(i)$ passes M_{m-1} to the input of $Unit(i)$

If a message labeled beyond $m-1$ is scheduled in the above, M_{m-1} is used instead. Note that if $n \neq n'$, an extra step is taken at the end of the pipeline phase for the input and output to exchange M_{m-2} and M_{m-1}. Therefore the total number of steps needed by this algorithm is $q + m = \lceil \log n \rceil + m - 1$, matching the lower bound on communication steps. Table 2 depicts an example of the schedule.

The correctness of the cooperation schedule can be proved by showing that at each pipeline step, every process has the message it needs to send according to the schedule. Here we sketch a proof by induction on pipeline step j for process i. The initial condition can be verified from the result of the setup phase and the initial role assignment. Now suppose the cooperation schedule is correct before step $j = \mathrm{J}$ of the pipeline phase. Without loss of generality, assume $i = Rep(i)$ and let $e = (\mathrm{J} \mod q)$, $f = ((\mathrm{J}+1) \mod q)$. During step J, we have:

- if $i_e = 1$, the output of $Unit(i)$ has M_{J} and the input receives $M_{\mathrm{J}+Dis[e]}$ from the partner unit. i and $Co(i)$ switch roles after the step.

Table 2. Cooperation schedule, $n = 6$, $m = 4$, cooperative process units are $\{0\}$, $\{1, 4\}$, $\{2, 5\}$, $\{3\}$. The top row gives $(i, Co(i), Rep(i))$ tuple for each process. The middle section shows the source and destination of each message during each step. The right section of the table shows messages received by every process after each step.

phase	step	M_0	M_1	M_2	M_3	(1,4,1)	(2,5,2)	(3,3,3)	(4,1,1)	(5,2,2)
setup	0	$0 \to 1$				M_0				
	1	$1 \to 3$	$0 \to 2$			M_0	M_1	M_0		
pipeline	0	$3 \to 5$	$2 \to 3$	$0 \to 4$		M_0	M_1	M_0, M_1	M_2	M_0
	1	$5 \to 2$ $1 \to 4$	$3 \to 1$	$4 \to 3$	$0 \to 5$	M_0, M_1	M_1, M_0	$M_0, M_1,$ M_2	M_2, M_0	M_0, M_3
	2		$1 \to 4$ $2 \to 5$	$3 \to 2$	$5 \to 3$ $0 \to 1$	$M_0, M_1,$ M_3	$M_1, M_0,$ M_2	$M_0, M_1,$ M_2, M_3	$M_2, M_0,$ M_1	$M_0, M_3,$ M_1
	3			$4 \to 1$ $2 \to 5$	$1 \to 4$ $5 \to 2$	$M_0, M_1,$ M_3, M_2	$M_1, M_0,$ M_2, M_3	$M_0, M_1,$ M_2, M_3	$M_2, M_0,$ M_1, M_3	$M_0, M_3,$ M_1, M_2

– if $i_e = 0$, the output of $Unit(i)$ has $M_{\mathrm{J}+Dis[e]}$ and the input receives M_{J} from the partner unit. There is no switch of roles after step J.

So before step J+1, the input of $Unit(i)$ has M_{J} that it needs to send during step $\mathrm{J}+1$. The output has $M_{\mathrm{J}+Dis[e]}$. During step $\mathrm{J}+1$, it needs to send out $M_{\mathrm{J}+1}$ or $M_{\mathrm{J}+1+Dis[f]}$, for $i_f = 1$ or $i_f = 0$ respectively. According to the construction of Dis:

– if $i_f = 0$, then $Dis[e] = Dis[f] + 1$;
– otherwise $Dis[e] = 1$.

In either case, the output has the message it needs to send during step $\mathrm{J}+1$. The correctness of the cooperation schedule follows.

5 Performance Studies

To evaluate the performance of the process cooperation algorithm, we have run experiments of MPI_BCAST implemented using the process cooperation algorithm and three other widely used algorithms: binomial tree, pipelined chain and pipelined binary tree. The experiment platform is a cluster of 32 IBM P5 [8] nodes connected by IBM HPS switch [7]. For each of the pipelined algorithms, we have experimented various m values and selected different m values that give the best performance for different data size ranges.

Figure 2 compares the MPI_BCAST implementations on 18 processes (left) and 32 processes (right) respectively, running one process per node, for data sizes up to 16MB. It can be seen that for large data sizes, the process cooperation algorithm outperforms the other algorithms by large margins: as much as 50% improvement in bandwidth over the pipelined chain algorithm, 80% over the pipelined binary tree algorithm and more than two times better than the binomial tree algorithm. Figure 3 shows latency of 256KB (left) and 4MB (right)

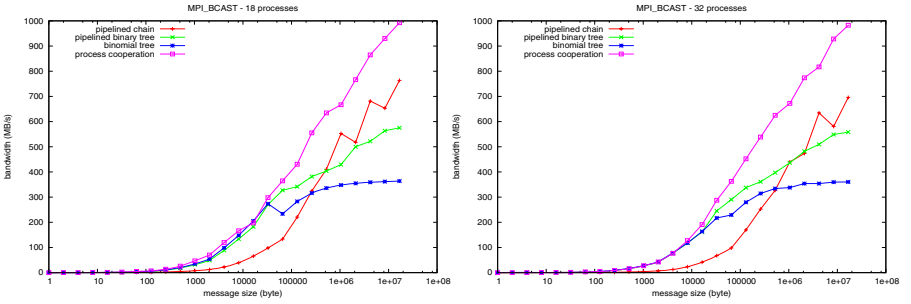


Fig. 2. Bandwidth comparison of MPI_BCAST, on 18 (left) and 32 (right) processes, data size up to 16MB

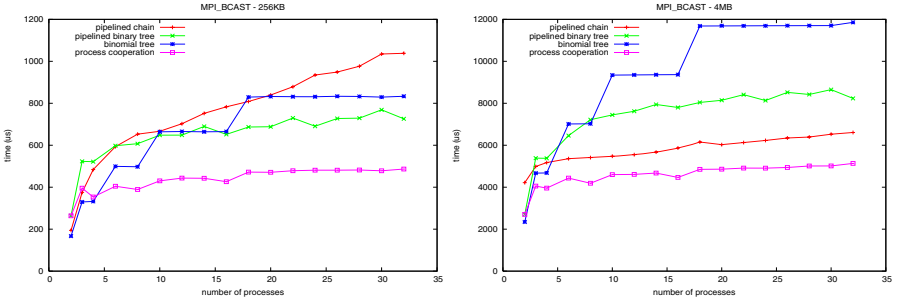


Fig. 3. Latency comparison of MPI_BCAST on 2 to 32 processes, data sizes: 256KB (left) and 4MB (right)

MPI_BCAST on varying number of processes (one per node). The process cooperation algorithm shows the best scaling as number of processes increases.

The overall results demonstrate performance trends similar to those reported in [12], which is expected since the communication schedules constructed by the process cooperation algorithm and the algorithm in [12] are both optimal. The $O(\log n)$ schedule construction of the process cooperation algorithm, however, is more efficient than the $O(n \times \log n)$ schedule construction in [12].

6 Conclusion and Future Studies

In this paper, we present a process cooperation algorithm for multiple message broadcast. This algorithm is optimal in one-port fully-connected systems. It provides a feasible and efficient solution to the practical implementation issue of on-line communication schedule construction in communication libraries, such as MPI. We have implemented and incorporated the process cooperation algorithm in IBM Parallel Environment/MPI.

On fully-connected system, the idea of process cooperation can be extended to multi-port communication models. We have developed new multi-port multiple

message broadcast algorithm that is theoretically better than known algorithms on arbitrary number of processes. Further performance study on multi-port broadcast will be conducted. We also plan to investigate the effectiveness of process cooperation in communication models that are more realistic for short messages.

References

1. Bar-Noy, A., Kipnis, S., Schieber, B.: Optimal multiple message broadcasting in telephone-like communication systems. In: Proceedings of the 6th Symposium on Parallel and Distributed Processing, IEEE, Los Alamitos (1996)
2. Barnett, M., Gupta, S., Payne, D., Shuler, L., van de Geijn, R., Watts, J.: Interprocessor collective communication library (intercom). In: Proceedings of the Scalable High Performance Computing Conference 1994 (1994)
3. Beaumont, O., Legrand, A., Marchal, L., Robert, Y.: Pipelining broadcasts on heterogeneous platforms. In: Proceedings of 18th IEEE International Parallel and Distributed Processing Symposium IPDPS04, IEEE Computer Society Press, Los Alamitos (2004)
4. Chan, E., Heimlich, M., Purkayastha, A., Van de Geijn, R.: On optimizing collective communication. In: Proceedings of 2004 IEEE International Conference on Cluster Computing, IEEE Computer Society Press, Los Alamitos (2004)
5. IBM PE for AIX 5L V4.3: MPI Programming Guide http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=/com.ibm.cluster.pe.doc/pebooks.html
6. Johnsson, S., Ho, C.T.: Optimum broadcasting and personalized communication in hypercubes. IEEE Transactions on Computers 38(9), 1249–1268 (1989)
7. Johnston, F., King-Smith, B.: Ibm pseries high performance switch. Technical report, IBM System and Technology Group (2006)
8. IBM System p5 UNIX servers www.ibm.com/systems/p/
9. Patarasuk, P., Faraj, A., Yuan, X.: Pipelined broadcast on ethernet swtiched clusters. In: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society Press, Los Alamitos (2006)
10. Sanders, P., Sibeyn, J.F.: A bandwidth latency tradeoff for broadcast and reduction. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 918–926. Springer, Heidelberg (2000)
11. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI – The Complete Reference, 2nd edn. The MPI Core, vol. 1. MIT Press, Cambridge (1998)
12. Träff, J., Ripke, A.: Optimal broadcast for fully connected networks. In: Yang, L.T., Rana, O.F., Di Martino, B., Dongarra, J.J. (eds.) HPCC 2005. LNCS, vol. 3726, pp. 45–56. Springer, Heidelberg (2005)
13. Worringen, J.: Pipelining and overlapping for mpi collective operations. In: 28th Annual IEEE International Conference for Local Computer Networks (LCN03), IEEE Computer Society Press, Los Alamitos (2003)

Self-consistent MPI Performance Requirements*

Jesper Larsson Träff[1], William Gropp[2], and Rajeev Thakur[2]

[1] NEC Laboratories Europe, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
traff@ccrl-nece.de
[2] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{gropp,thakur}@mcs.anl.gov

Abstract. The MPI Standard does not make any performance guarantees, but users expect (and like) MPI implementations to deliver good performance. A common-sense expectation of performance is that an MPI function should perform no worse than a combination of other MPI functions that can implement the same functionality. In this paper, we formulate some performance requirements and conditions that good MPI implementations can be expected to fulfill by relating aspects of the MPI standard to each other. Such a performance formulation could be used by benchmarks and tools, such as *SKaMPI* and *Perfbase*, to automatically verify whether a given MPI implementation fulfills basic performance requirements. We present examples where some of these requirements are not satisfied, demonstrating that there remains room for improvement in MPI implementations.

1 Introduction

For good reasons MPI (the *Message Passing Interface*) [4,9] comes without a performance model and, apart from some "advice to implementers," without any requirements or recommendations as to what a good implementation should satisfy regarding performance. The main reasons are, of course, that the implementability of the MPI standard should not be restricted to systems with specific interconnect capabilities and that implementers should be given maximum freedom in how to realize the various MPI constructs. The widespread use of MPI over an extremely wide range of systems, as well as the many existing and quite different implementations of the standard, show that this was a wise decision.

On the other hand, for the analysis and performance prediction of applications, a performance model is needed. Abstract models such as LogP [3] and

* This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 36–45, 2007.

BSP [13] tend to be too complex (for full applications), too limited, or too abstract to have predictive power. MPI provides much more flexible (but also much more complex) modes of communication than catered to in these models.

An alternative is to use MPI itself as a model and analyze applications in terms of certain basic MPI primitives. This may work well for restricted usages of MPI to, say, the MPI collectives, but full MPI is probably too large to be a tractable model for performance analysis and prediction.

A related consideration is a hard-to-quantify desire for *performance portability*—the desire that an application should, in some qualitative sense, behave the same when ported to a new system or linked with a different MPI library. Detailed, public benchmarks of MPI constructs can help in translating the performance of an application on one system and MPI library to another system with another MPI library [8]. Accurate performance models would also facilitate translation between systems and MPI libraries, but in their absence simple MPI-intrinsic requirements to MPI implementations might serve to guard against the most unpleasant surprises.

MPI has many ways of expressing the same communication (patterns), with varying degrees of generality and freedom for the application programmer. This kind of universality makes it possible to relate aspects of MPI to each other also in terms of the expected performance. This is utilized already in the MPI definition itself, where certain MPI functions are explained in a semi-formal way in terms of other MPI functions.

The purpose of this paper is to discuss whether it is possible, sensible, and desirable to formulate system-independent, but MPI-intrinsic performance requirements that a "good" MPI implementation should fulfill. Such requirements should not make any commitments to particular system capabilities but would enforce a high degree of performance consistency of an MPI implementation. For example, similar optimizations would have to be done for collective operations that are interlinked through such performance rules. Furthermore, such rules, even if relatively trivial, would provide a kind of "sanity check" of an MPI implementation, especially if they could be checked automatically. In this paper, we formulate a number of MPI-intrinsic performance requirements by semi-formally relating different aspects of the MPI standard to each other, which we refer to as *self-consistent performance requirements*. By their very nature the rules can be used only to ensure *consistency*—a trivial, bad MPI implementation could fulfill them as well as a carefully tuned library.

Related work includes quality of service for numerical library components [5,6]. Because of the complexity of these components, it is not possible to provide the sort of definitive ordering that we propose for MPI communications.

2 General Rules and Notation

We first formulate and discuss a number of general self-consistent MPI performance requirements, presupposing reasonable familiarity with MPI. We consider the following relationships (metarules) between MPI routines:

1. Replacing all communication with the appropriate use of `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` should not reduce the communication time. In the context of MPI-2, this can be applied even to the MPI one-sided communication routines.
2. Subdividing messages into multiple messages should not reduce the communication time.
3. Replacing a routine with a similar routine that provides additional semantic guarantees should not reduce the communication time.
4. For collective routines, replacing a collective routine with several routines should not reduce the communication time. In particular, the specification of most of the MPI collective routines includes a description in terms of other MPI routines; each MPI collective should be at least as fast as that description.
5. For process topologies, communicating with a communicator that implements a process topology should not be slower than using a random communicator.

The first of these requirements provides a formal way to derive relationships between the MPI communication routines—write each routine in terms of an equivalent use of `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait`, and then compare the time taken. In the rest of this paper, we give more specific examples of each of these rules.

We use the notation that

$$\texttt{MPI\_}A(n) \preceq \texttt{MPI\_}B(n) \tag{1}$$

means that MPI functionality A is not slower than B when evoked with parameters resulting in the same amount of communication or computation n. Note that MPI buffers are not always specified in this way. We use p to denote the number of processes involved in a call, and $\texttt{MPI\_}A\{c\}$ for A called on communicator c. As an example

$$\texttt{MPI\_Send}(n) \preceq \texttt{MPI\_Isend}(n) + \texttt{MPI\_Wait} \tag{2}$$

states that an `MPI_Send` call is possibly faster, but at least not slower than a call to `MPI_Isend` with the same parameters followed by an `MPI_Wait` call. In this case, it would probably make sense to require more strongly that

$$\texttt{MPI\_Send}(n) \approx \texttt{MPI\_Isend}(n) + \texttt{MPI\_Wait} \tag{3}$$

which means that the alternatives perform similarly. Quantifying the meaning of "similarly" is naturally contentious. A strong definition would say that there is a small confidence interval such that for any data size n, the running time of the one construct is within the running time of the other with this confidence interval. A somewhat weaker definition could require that the running time of the two constructs is within a small constant factor of each other for any data size n.

3 General Communication

Rule 2 can be made more precise as follows: Splitting a communication buffer of kn units into k buffers of n units, and communicating separately, never pays off.

$$\texttt{MPI\_}A(kn) \preceq \underbrace{\texttt{MPI\_}A(n) + \cdots + \texttt{MPI\_}A(n)}_{k} \tag{4}$$

For an example where this rule is violated with $A = \texttt{Bcast}$, see [1, p. 68].

Similarly, splitting possibly structured data into its constituent blocks of fixed size k should also not be faster.

$$\texttt{MPI\_}A(kn) \preceq \underbrace{\texttt{MPI\_}A(k) + \cdots + \texttt{MPI\_}A(k)}_{n} \tag{5}$$

One might be able to elaborate this requirement into a formal requirement for the performance of user-defined MPI datatypes, but this issue would require much care.

Note that many MPI implementations will violate Rule 2 and (4) because of the use of eager and rendezvous message protocols. An example is shown in Figure 1. A user with a 1500-byte message will achieve better performance on this system by sending two 750-byte messages. This example shows one of the implementation features that competes with performance portability—in this case, the use of limited message buffers. To satisfy Rule 2, an MPI implementation would need a more sophisticated buffer management strategy, but in turn this could decrease the performance of all short messages.

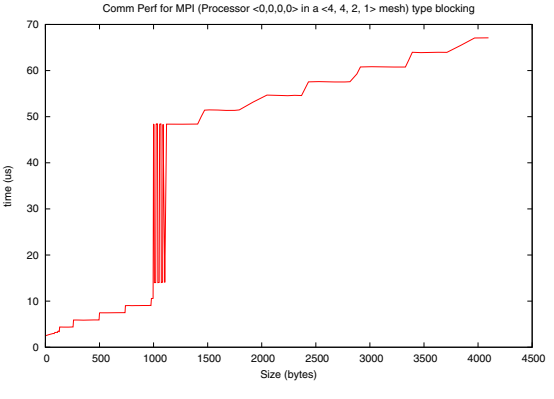


Fig. 1. Measured performance of short messages on IBM BG/L. Note the large jump around 1024 bytes; this is the transition from eager to rendezvous protocol in the MPI implementation.

As an example of Rule 3, we have the following

$$\texttt{MPI\_Send} \preceq \texttt{MPI\_Ssend} \tag{6}$$

Since the synchronous send routine has an additional semantic guarantee (the routine cannot return until the matching receive has started), it should not be faster than the regular send.

4 Collective Communication

The MPI collectives are strongly interrelated semantically, and often one collective can be implemented in terms of one or more other related collectives. A general requirement (metarule) is that a specialized collective should not be slower than a more general collective. Thus, with good conscience, users can be given the advice to always use the most specific collective (which is, of course, exactly the motivation for having so many collectives in MPI).

4.1 Regular Communication Collectives

The following three rules are instances of the metarule that specialized functions should not be slower than more general ones.

$$\texttt{MPI\_Gather}(n) \preceq \texttt{MPI\_Allgather}(n) \tag{7}$$

$$\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Allgather}(n) \tag{8}$$

$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Alltoall}(n) \tag{9}$$

The next rule implements a collective operation in terms of two others. Again the specialized function (`MPI_Allgather`) should not be slower.

$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Gather}(n) + \texttt{MPI\_Bcast}(n) \tag{10}$$

This is not as trivial as it may look. If, for instance, a linear ring algorithm is used for the `MPI_Allgather`, but tree-based algorithms for `MPI_Gather` and `MPI_Bcast`, the relationship will not hold (at least for small n).

A less obvious requirement relates `MPI_Scatter` to `MPI_Bcast`. The idea is to implement the `MPI_Scatter` function, which scatters individual data to each of the p processes, by a broadcast of the combined data of size pn; each process copies out its block from the larger buffer.

$$\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Bcast}(pn) \tag{11}$$

Again this is a nontrivial requirement for small n for MPI libraries with an efficient `MPI_Bcast` implementation and forces an equally efficient implementation of `MPI_Scatter`.

A currently popular implementation of broadcast for large messages is by a scatter followed by an allgather operation [2,10]. Since this is an algorithm expressed purely in terms of collective operations, it makes sense to require that the native broadcast operation should behave at least as well.

$$\texttt{MPI\_Bcast}(n) \preceq \texttt{MPI\_Scatter}(n) + \texttt{MPI\_Allgather}(n) \tag{12}$$

4.2 Reduction Collectives

The second half of the next rule states that a good MPI implementation should have an `MPI_Allreduce` that is faster than the trivial implementation of reduction to root followed by a broadcast.

$$\begin{aligned} \texttt{MPI\_Reduce}(n) &\preceq \texttt{MPI\_Allreduce}(n) \\ &\preceq \texttt{MPI\_Reduce}(n) + \texttt{MPI\_Bcast}(n) \end{aligned} \tag{13}$$

A similar rule can be formulated for `MPI_Reduce_scatter`.

$$\begin{aligned} \texttt{MPI\_Reduce}(n) &\preceq \texttt{MPI\_Reduce\_scatter}(n) \\ &\preceq \texttt{MPI\_Reduce}(n) + \texttt{MPI\_Scatterv}(n) \end{aligned} \tag{14}$$

The next two rules implement `MPI_Reduce` and `MPI_Allreduce` in terms of `MPI_Reduce_scatter` and are similar to the broadcast implementation of requirement (12).

$$\texttt{MPI\_Reduce}(n) \preceq \texttt{MPI\_Reduce\_scatter}(n) + \texttt{MPI\_Gather}(n) \tag{15}$$

$$\texttt{MPI\_Allreduce} \preceq \texttt{MPI\_Reduce\_scatter}(n) + \texttt{MPI\_Allgather}(n) \tag{16}$$

For the reduction collectives, MPI provides a set of built-in binary operators, as well as the possibility for users to define their own operators. A natural requirement is that a user-defined implementation of a built-in operator should not be faster.

$$\texttt{MPI\_Reduce}(\texttt{MPI\_SUM}) \preceq \texttt{MPI\_Reduce}(\texttt{user\_sum}) \tag{17}$$

A curious example where this is violated is again given in [1, p. 65]. For a particular vendor MPI implementation, a user-defined sum operation was significantly faster than the built-in `MPI_SUM` operation!

4.3 Irregular Communication Collectives

The irregular collectives of MPI, in which the amount of data communicated between pairs of processes may differ, are obviously more general than their regular counterparts. It is desirable that performance be similar when an irregular collective is used to implement the functionality of the corresponding regular collective. Thus, we have requirements like the following.

$$\texttt{MPI\_Gatherv}(v) \approx \texttt{MPI\_Gather}(n) \tag{18}$$

This requires that the performance of `MPI_Gatherv` be in the same ballpark as the regular `MPI_Gather` for uniform p element vectors v with $v[i] = n/p$. Again this is not a trivial requirement. For instance, there are easy tree-based algorithms for `MPI_Gather` but not for `MPI_Gatherv` (at least not as easy because the irregular counts are not available on all processes), and thus performance characteristics of the two collectives may be quite different [11]. Thus, $\approx$ should be formulated carefully and leave room for some overhead.

4.4 Constraining Implementations

In addition to the rule above that relates collective operations to other collective operations, it would be tempting to require that a good MPI implementation fulfill some minimal requirements regarding the performance of its collectives. For example, the MPI standard already explains many of the collectives in terms of send and receive operations.

$$\texttt{MPI\_Gather}(n) \preceq \underbrace{\texttt{MPI\_Recv}(n/p) + \cdots + \texttt{MPI\_Recv}(n/p)}_{p} \tag{19}$$

Extending this, one could define a set of "minimal implementations," for example, an `MPI_Bcast` implementation by a simple binomial tree. Correspondingly one could require that the collectives of an MPI library perform at least as well. This requirement could prevent trivial implementations from fulfilling the rules, but how far this idea could and should be taken is not clear at present.

5 Communicators and Topologies

Let c be a communicator (set of processes) of size p representing an assignment of p processes to p processors. Let c' be a communicator representing a different (random) assignment to the same processors. A metarule like

$$\texttt{MPI\_}A\{c\} \approx \texttt{MPI\_}A\{c'\} \tag{20}$$

can be expected to hold for homogeneous systems for any MPI communication operation A. For non-homogeneous systems, such as SMP clusters with a hierarchical communication system, such a rule will not hold.

For some collectives, it is still reasonable to require communicator independence (irrespective of system), for example, the following.

$$\texttt{MPI\_Allgather}\{c\} \approx \texttt{MPI\_Allgather}\{c'\} \tag{21}$$

This is not a trivial requirement. A linear ring or logarithmic algorithm designed on the assumption of a homogeneous system may, when executed on a SMP system and depending on the distribution of the MPI processes over the SMP nodes, have communication rounds in which more than one MPI process per SMP node communicates with processes on other nodes. The effect of the resulting serialization of communication is shown in Figure 2.

It seems reasonable to require communicator independence for all symmetric (non-rooted communication) collectives, that is, requirement (20) for $A \in \{\texttt{Allgather}, \texttt{Alltoall}, \texttt{Barrier}\}$.

MPI contains routines for defining process topologies, and these should not decrease performance for their preferred communication patterns. As an example of Rule 5, using a Cartesian communicator c and then communicating to the Cartesian neighbors should be no slower than using an arbitrary communicator c'.

$$\texttt{MPI\_Send}\{c\} \preceq \texttt{MPI\_Send}\{c'\} \tag{22}$$

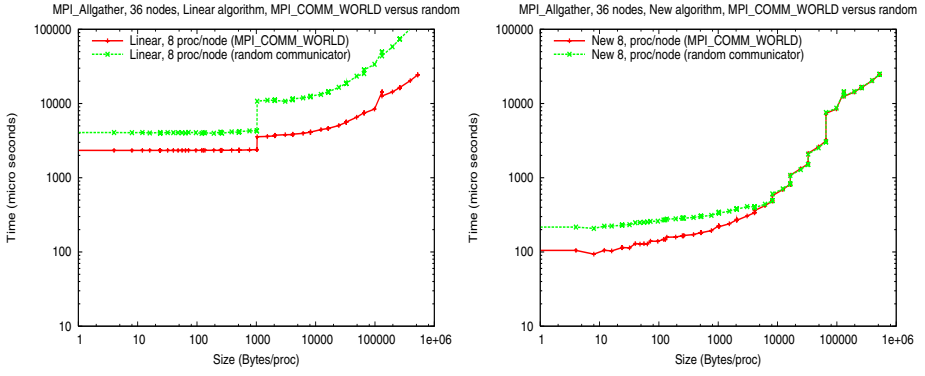


Fig. 2. Left: performance of a simple, non-SMP-aware linear ring algorithm for `MPI_Allgather` when executed on the ordered `MPI_COMM_WORLD` communicator and on a communicator where the processes have been randomly permuted. Right: performance of an SMP-aware algorithm for `MPI_Allgather` on ordered and random communicator. The degradation for small data for the random communicator is due to specifics of the target (vector) system; see [12].

6 One-Sided Communication

The one-sided communication model of MPI-2 is interrelated to both point-to-point and collective communication, and a number of performance requirements can be formulated. We give a single example. For the fence synchronization method, the performance of a fence-put-fence should be no worse than a barrier on the same communicator, followed by an `MPI_Send` of the datatype and address information, followed by another barrier:

$$\texttt{MPI\_Win\_fence} + \texttt{MPI\_Put}(n) + \texttt{MPI\_Win\_fence} \preceq \texttt{MPI\_Barrier} + \texttt{MPI\_Send}(d) + \texttt{MPI\_Send}(n) + \texttt{MPI\_Barrier} \tag{23}$$

where d represents information about the address and datatype on the target.

Figure 3 shows an example where this requirement is violated. On an IBM SMP system with IBM's MPI, the performance of a simple nearest-neighbor halo exchange is about three times worse with one-sided communication and fence synchronization compared with regular point-to-point communication, even when two processors are available for each MPI process.

7 Automating the Checks

With a precise definition of the $\preceq$ and $\approx$ relations, it would in principle be possible to automate the checking that a given MPI implementation (on a given system) fulfills a set of self-consistent performance requirements. A customizable benchmark such as *SKaMPI* [1,7,8] already has some patterns that allow the comparison

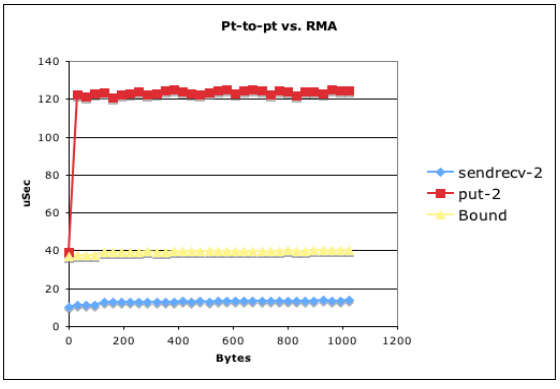


Fig. 3. Performance of `MPI_Put` with fence synchronization versus point-to-point communication on an IBM SMP with IBM's MPI for a simple nearest-neighbor halo exchange. The middle line shows the performance bound based on using barrier and send.

of alternative implementations of the same MPI functionality, similar to many of the rules formulated above. It would be easy to incorporate a wider set of rules into *SKaMPI*. By combining this with an experiments-management system such as *Perfbase* [14,15], one could create a tool that automatically validates an MPI implementation as to its intrinsic performance. (We have not done so yet.)

8 Concluding Remarks

Users often complain about the poor performance of some of the MPI functions in MPI implementations and of the difficulty of writing code whose performance is portable. Solving this problem requires defining performance standards that MPI implementations are encouraged to follow. We have defined some basic, intrinsic performance rules for MPI implementations and provided examples where some of these rules are being violated. Further experiments might reveal more such violations. We note that just satisfying these rules does not mean that an implementation is good, because even a poor, low-quality implementation can trivially do so. They must be used in conjunction with other benchmarks and performance metrics for a comprehensive performance evaluation of MPI implementations.

References

1. Augustin, W., Worsch, T.: Usefulness and usage of SKaMPI-bench. In: Dongarra, J.J., Laforenza, D., Orlando, S. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2840, pp. 63–70. Springer, Heidelberg (2003)
2. Barnett, M., Gupta, S., Payne, D.G., Schuler, L., van de Geijn, R., Watts, J.: Building a high-performance collective communication library. In: Supercomputing'94, pp. 107–116 (1994)

3. Culler, D.E., Karp, R.M., Patterson, D., Sahay, A., Santos, E.E., Schauser, K.E., Subramonian, R., von Eicken, T.: LogP: A practical model of parallel computation. Communications of the ACM 39(11), 78–85 (1996)
4. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI – The Complete Reference. The MPI Extensions, vol. 2. MIT Press, Cambridge (1998)
5. McInnes, L.C., Ray, J., Armstrong, R., Dahlgren, T.L., Malony, A., Norris, B., Shende, S., Kenny, J.P., Steensland, J.: Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration. Technical Report ANL/MCS-P1326-0206, Argonne National Laboratory (February 2006)
6. Norris, B., McInnes, L., Veljkovic, I.: Computational quality of service in parallel CFD. In: Proceedings of the 17th International Conference on Parallel Computational Fluid Dynamics, University of Maryland, College Park, MD, May 24–27 (to appear, 2006)
7. Reussner, R., Sanders, P., Prechelt, L., Müller, M.: SKaMPI: A detailed, accurate MPI benchmark. In: Alexandrov, V.N., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1497, pp. 52–59. Springer, Heidelberg (1998)
8. Reussner, R., Sanders, P., Träff, J.L.: SKaMPI: A comprehensive benchmark for public benchmarking of MPI. Scientific Programming 10(1), 55–65 (2002)
9. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI – The Complete Reference, 2nd edn. The MPI Core, vol. 1. MIT Press, Cambridge (1998)
10. Thakur, R., Gropp, W.D., Rabenseifner, R.: Improving the performance of collective operations in MPICH. International Journal on High Performance Computing Applications 19, 49–66 (2004)
11. Träff, J.L.: Hierarchical gather/scatter algorithms with graceful degradation. In: International Parallel and Distributed Processing Symposium (IPDPS 2004), p. 80 (2004)
12. Träff, J.L.: Efficient allgather for regular SMP-clusters. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 58–65. Springer, Heidelberg (2006)
13. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM 33(8), 103–111 (1990)
14. Worringen, J.: Experiment management and analysis with perfbase. In: IEEE International Conference on Cluster Computing (2005)
15. Worringen, J.: Automated performance comparison. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 402–403. Springer, Heidelberg (2006)

Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE

Rajeev Thakur and William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{thakur,gropp}@mcs.anl.gov

Abstract. MPI implementations that support the highest level of thread safety for user programs, `MPI_THREAD_MULTIPLE`, are becoming widely available. Users often expect that different threads can execute independently and that the MPI implementation can provide the necessary level of thread safety with only a small overhead. The MPI Standard, however, requires only that no MPI call in one thread block MPI calls in other threads; it makes no performance guarantees. Therefore, some way of measuring an implementation's performance is needed. In this paper, we propose a number of performance tests that are motivated by typical application scenarios. These tests cover the overhead of providing the `MPI_THREAD_MULTIPLE` level of thread safety for user programs, the amount of concurrency in different threads making MPI calls, the ability to overlap communication with computation, and other features. We present performance results with this test suite on several platforms (Linux cluster, Sun and IBM SMPs) and MPI implementations (MPICH2, Open MPI, IBM, and Sun).

1 Introduction

With thread-safe MPI implementations becoming increasingly common, users are able to write multithreaded MPI programs that make MPI calls concurrently from multiple threads. Thread safety does not come for free, however, because the implementation must protect certain data structures or parts of the code with mutexes or critical sections. Developing a thread-safe MPI implementation is a fairly complex task, and the implementers must make several design choices, both for correctness and for performance [2]. To simplify the task, implementations often focus on correctness first and performance later (if at all). As a result, even though an MPI implementation may support multithreading, its performance may be far from optimized. Users, therefore, need a way to determine how efficiently an implementation can support multiple threads. Similarly, as implementers experiment with a potential performance optimization, they need a way to measure the outcome. (We ourselves face this situation in MPICH2.) To meet these needs, we have created a test suite that can shed light

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 46–55, 2007.

on the performance of an MPI implementation in the multithreaded case. We describe the tests in the suite, the rationale behind them, and their performance with several MPI implementations (MPICH2, Open MPI, IBM MPI, and Sun MPI) on several platforms.

Related Work. The MPI benchmarks from Ohio State University [4] contain a multithreaded latency test, which is a ping-pong test with one thread on the sender side and two (or more) threads on the receiver side. A number of other MPI benchmarks exist, such as SKaMPI [6] and the Intel MPI Benchmarks [3], but they do not measure the performance of multithreaded MPI programs. A good discussion of the issues in developing a thread-safe MPI implementation is given in [2]. Other thread-safe MPI implementations are described in [1,5].

2 Overview of MPI and Threads

To understand the test suite and the rationale behind each test, one must understand the thread-safety specification in MPI. For performance reasons, MPI defines four "levels" of thread safety and allows the user to indicate the level desired—the idea being that the implementation need not incur the cost for a higher level of thread safety than the user needs. The four levels of thread safety are as follows:

1. `MPI_THREAD_SINGLE` Each process has a single thread of execution.
2. `MPI_THREAD_FUNNELED` A process may be multithreaded, but only the thread that initialized MPI may make MPI calls.
3. `MPI_THREAD_SERIALIZED` A process may be multithreaded, but only one thread at a time may make MPI calls.
4. `MPI_THREAD_MULTIPLE`A process may be multithreaded, and multiple threads may simultaneously call MPI functions (with some restrictions mentioned below).

An implementation is not required to support levels higher than `MPI_THREAD_SINGLE`; that is, an implementation is not required to be thread safe. A fully thread-compliant implementation, however, will support `MPI_THREAD_MULTIPLE`. MPI provides a function, `MPI_Init_thread`, by which the user can indicate the level of thread support desired, and the implementation will return the level supported. A portable program that does not call `MPI_Init_thread` should assume that only `MPI_THREAD_SINGLE` is supported. The tests described in this paper focus on the `MPI_THREAD_MULTIPLE` (fully multithreaded) case.

For `MPI_THREAD_MULTIPLE`, the MPI Standard specifies that when multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order. Also, blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions. MPI also says that it is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls. For example,

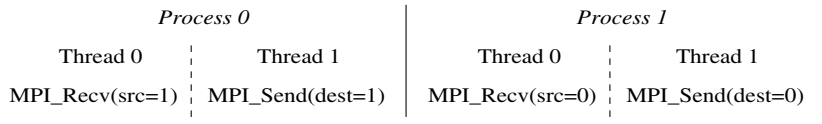


Fig. 1. An implementation must ensure that this example never deadlocks for any ordering of thread execution

the user cannot call `MPI_Info_set` and `MPI_Info_free` on the same `info` object concurrently from two threads of the same process; the user must ensure that the `MPI_Info_free` is called only after `MPI_Info_set` returns on the other thread. Similarly, the user must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads.

A straightforward implication of the MPI thread-safety specification is that an implementation cannot implement thread safety by simply acquiring a lock at the beginning of each MPI function and releasing it at the end of the function: A blocked function that holds a lock may prevent MPI functions on other threads from executing, a situation that in turn might prevent the occurrence of the event that is needed for the blocked function to return. An example is shown in Figure 1. If thread 0 happened to get scheduled first on both processes, and `MPI_Recv` simply acquired a lock and waited for the data to arrive, the `MPI_Send` on thread 1 would not be able to acquire its lock and send its data; hence, the `MPI_Recv` would block forever. Therefore, the implementation must release the lock at least before blocking within the `MPI_Recv` and then reacquire the lock if needed after the data has arrived. (The tests described in this paper provide some information about the fairness and granularity of how blocking MPI functions are handled by the implementation.)

3 The Test Suite

Users of threads in MPI often have the following expectations of the performance of threads, both those making MPI calls and those performing computation concurrently with threads that are making MPI calls.

- The cost of thread safety, compared with lower levels of thread support, such as `MPI_THREAD_FUNNELED`, is relatively low.
- Multiple threads making MPI calls, such as `MPI_Send` or `MPI_Bcast`, can make progress simultaneously.
- A blocking MPI routine in one thread does not consume excessive CPU resources while waiting.

Our tests are designed to test these expectations; in terms of the above categories, they are as follows:

Cost of thread safety. One simple test to measure `MPI_THREAD_MULTIPLE` overhead.

Concurrent progress. Tests to measure concurrent bandwidth by multiple threads of a process to multiple threads of another process, as compared with multiple processes to multiple processes. Both point-to-point and collective operations are included.

Computation overlap. Tests to measure the overlap of communication with computation and the ability of the application to use a thread to provide a nonblocking version of a communication operation for which there is no corresponding MPI call, such as nonblocking collectives or I/O operations that involve several steps.

We describe the tests below and present performance results on the following platforms and MPI implementations:

Linux Cluster. We used the Breadboard cluster at Argonne, in which each node has two dual-core 2.8 GHz AMD Opteron CPUs. The nodes are connected by Gigabit Ethernet. We used MPICH2 1.0.5 and Open MPI 1.2.1.

Sun Fire SMP. We used a Sun Fire SMP from the Sun cluster at the RWTH Aachen University. The specific machine we ran on was a Sun Fire E2900 with eight dual-core UltraSPARC IV 1.2 GHz CPUs. It runs Sun's MPI (ClusterTools 5).

IBM SMP. We also used an IBM p655+ SMP from the DataStar cluster at the San Diego Supercomputer Center. The machine has eight 1.7 GHz POWER4+ CPUs and runs IBM's MPI.

3.1 MPI_THREAD_MULTIPLE Overhead

Our first test measures the ping-pong latency for two cases of a *single*-threaded program: initializing MPI with just `MPI_Init` and initializing it with `MPI_Init_thread` for `MPI_THREAD_MULTIPLE`. This test demonstrates the overhead of ensuring thread safety for `MPI_THREAD_MULTIPLE`—typically implemented by acquiring and releasing locks—even though no special steps are needed in this case because the process is single threaded (but the implementation does not know that).

Figure 2 shows the results. On the Linux cluster, with both MPICH2 and Open MPI, the overhead of `MPI_THREAD_MULTIPLE` is less than 0.5 μs. On the IBM SMP with IBM MPI, it is less than 0.25 μs. On the other hand, on the Sun SMP with Sun MPI, the overhead is very high—more than 3 μs.

3.2 Concurrent Bandwidth

The second test measures the cumulative bandwidth obtained when multiple threads of a process communicate with multiple threads of another process compared with multiple processes instead of threads (see Figure 3). It demonstrates how much thread locks affect the cumulative bandwidth; ideally, the multiprocess and multithreaded cases should perform similarly.

Figure 4 shows the results. On the Linux cluster, the tests were run on two nodes, with all communication happening across nodes. We ran two cases: one

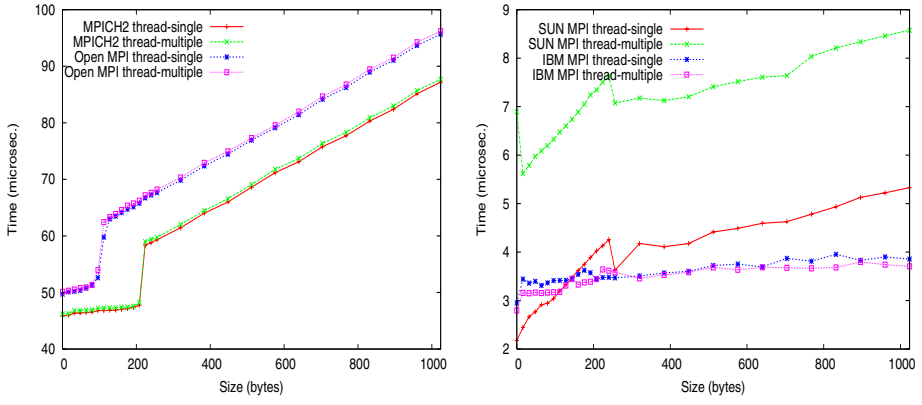


Fig. 2. Overhead of `MPI_THREAD_MULTIPLE` on the Linux cluster (left) and Sun and IBM SMPs (right)

where there were as many processes/threads as the number of processors on a node (four) and one where there were eight processes/threads running on four processors. In both cases, there is no measurable difference in bandwidth between threads and processes with MPICH2. With Open MPI, there is a decline in bandwidth with threads in the oversubscribed case.

On the Sun and IBM SMPs, on the other hand, there is a substantial decline (more than 50% in some cases) in the bandwidth when threads were used instead of processes. Although it is harder to provide low overhead in these shared-memory environments because the communication bandwidths are so high, we believe a tuned implementation should be able to support concurrent threads with a much smaller loss in performance.

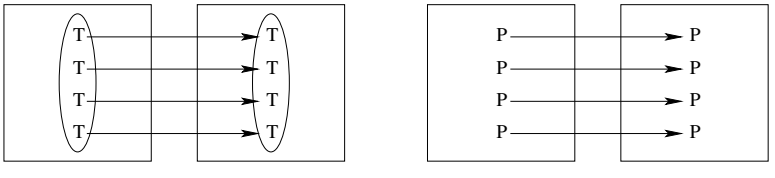


Fig. 3. Communication test when using multiple threads (left) versus multiple processes (right)

3.3 Concurrent Latency

Our third test is similar to the concurrent bandwidth test except that it measures the time for individual short messages instead of concurrent bandwidth for large messages. Figure 5 shows the results. On the Linux cluster with MPICH2, there is a 20 μs overhead in latency when using concurrent threads instead of processes. With Open MPI, the overhead is about 30 μs. With Sun and IBM MPI, the latency with threads is about 10 times the latency with processes.

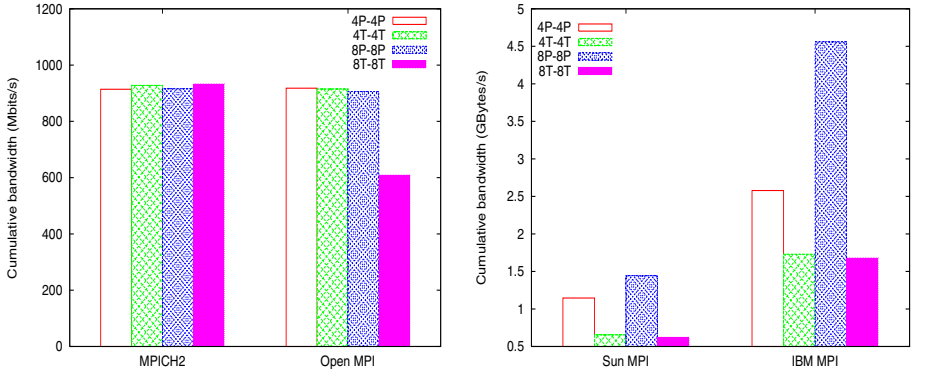


Fig. 4. Concurrent bandwidth test on Linux cluster (left) and Sun and IBM SMPs (right)

Again, although it is more difficult to provide low overhead on these machines because the basic message-passing latency is so low, a tuned implementation should be able to do better than a factor of 10 higher latency in the threaded case.

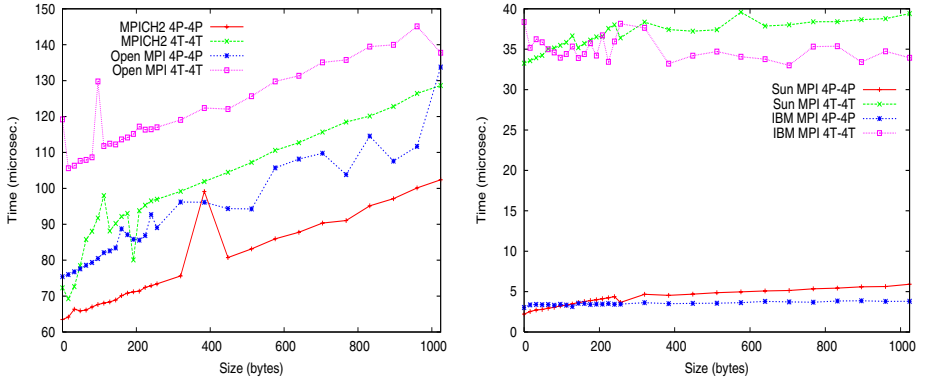


Fig. 5. Concurrent latency test on Linux cluster (left) and Sun and IBM SMPs (right)

3.4 Concurrent Short-Long Messages

The fourth test is a blend of the concurrent bandwidth and concurrent latency tests. It has two versions. In the threads version, rank 0 has two threads: one sends a long message to rank 1, and the other sends a series of short messages to rank 2. The second version of the test is similar except that the two senders are processes instead of threads. This test tests the fairness of thread scheduling and locking. If they were fair, one would expect each of the short messages to take roughly the same amount of time.

The results are shown in Figure 6. With both MPICH2 and Open MPI, the cost of communicating the long message is evenly distributed among a number

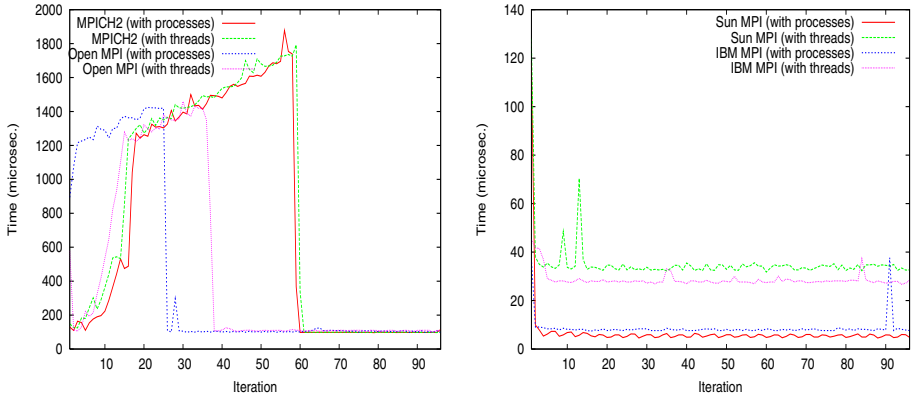


Fig. 6. Concurrent short-long messages test on Linux cluster (left) and Sun and IBM SMPs (right)

of short messages. A single short message is not penalized for the entire time the long message is communicated. This result demonstrates that, in the threaded case, locks are fairly held and released and that the thread blocked in the long-message send does not block the other thread. With Sun and IBM MPI, however, one sees spikes in the graphs. This behavior may be because these implementations use memory copying to communicate data, and it is harder to overlap this memory-copy time with the memory copying on the other thread.

3.5 Computation/Communication Overlap

Our fifth test measures the ability of an implementation to overlap communication with computation and provides users an alternative way of achieving such an overlap if the implementation does not do so. The test has two versions. The first version has an iterative loop in which a process communicates with its four nearest neighbors by posting nonblocking sends and receives, followed by a computation phase, followed by an `MPI_Waitall` for the communication to complete. The second version is similar except that, before the iterative loop, each process spawns a thread that blocks on an `MPI_Recv`. The matching `MPI_Send` is called by the main thread only at the end of the program, just before `MPI_Finalize`. The thread thus blocks in the `MPI_Recv` while the main thread is in the communication-computation loop. Since the thread is executing an MPI function, whenever it gets scheduled by the operating system, it can cause progress to occur on the communication initiated by the main thread. This technique effectively simulates asynchronous progress by the MPI implementation. If the total time taken by the communication-computation loop in this case is less than that in the nonthreaded version, it indicates that the MPI implementation on its own does not overlap communication with computation.

Figure 7 shows the results. Here, "no overlap" refers to the test without the thread, and "overlap" refers to the test with the thread. The results with MPICH2 demonstrate no asynchronous progress, as the overlap version of the

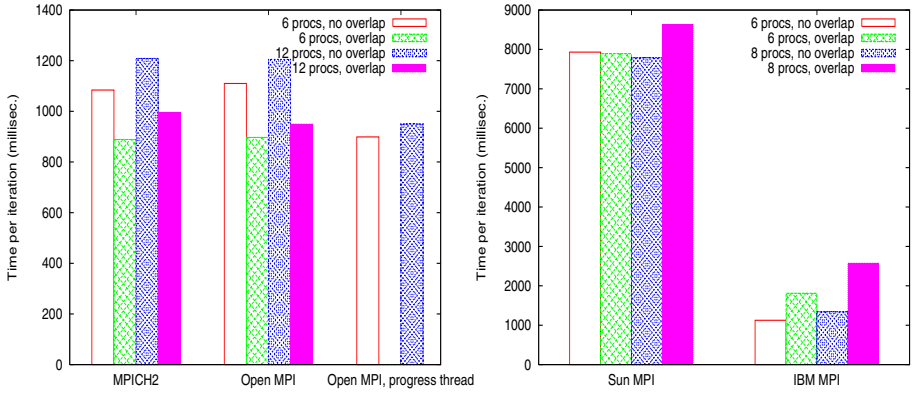


Fig. 7. Computation/communication overlap test on Linux cluster (left) and Sun and IBM SMPs (right)

test performs better. With Open MPI, we ran two experiments. We first used the default build; the results indicate that it performs similarly to MPICH2—no overlap of computation and communication. Open MPI can also be optionally built to use an extra thread internally for asynchronous progress. With this version of the library, we see that indeed there is asynchronous progress, as the performance is nearly the same as for the "overlap" test with the default build. That is, the case with the implementation-created progress thread performs similarly to the case with the user-created thread.

We note that always using an extra thread for progress has other performance implications. For example, it can result in higher communication latencies because of the thread-switching overhead. Due to lack of space, we did not run all the other tests with the version of Open MPI configured to use an extra thread.

The results on the Sun and IBM SMPs indicate no overlap. In fact, with eight processes, the performance was worse with the overlap thread because of the high overhead when using threads with these MPI implementations.

3.6 Concurrent Collectives

Our sixth test compares the performance of concurrent calls to a collective function (`MPI_Allreduce`) issued from multiple threads to that when issued from multiple processes. The test uses multiple communicators, and processes are arranged such that the processes belonging to a given communicator are located on different nodes. In other words, collective operations are issued by multiple threads/processes on a node, with all communication taking place across nodes (similar to Figure 3 but for collectives and using multiple nodes).

Figure 8 (left) shows the results on the Linux cluster. MPICH2 has relatively small overhead for the threaded version, compared with Open MPI.

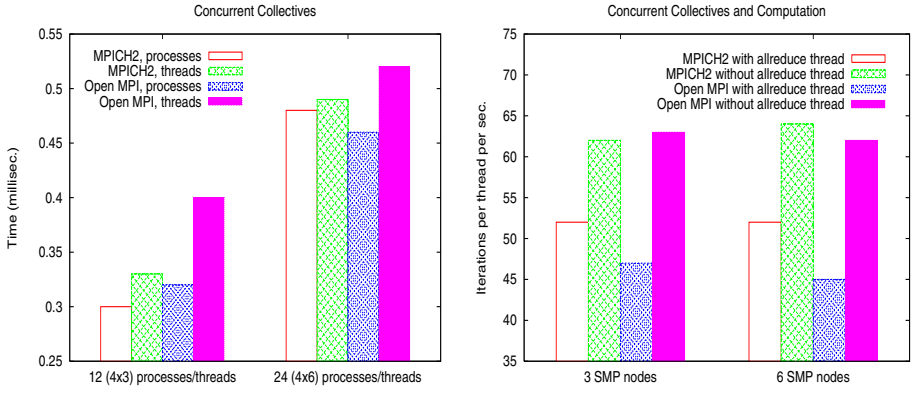


Fig. 8. Left: Concurrent collectives test on the Linux cluster (4x3 refers to 4 process/threads each on 3 nodes). Right: Concurrent collectives and computation test on the Linux cluster.

3.7 Concurrent Collectives and Computation

Our final test evaluates the ability to use a thread to hide the latency of a collective operation while using all available processors to perform computations. It uses $p+1$ threads on a node with p processors. Threads 0–$(p-1)$ perform some computation iteratively. Thread p does an `MPI_Allreduce` with its corresponding threads on other nodes. When the allreduce completes, it sets a flag, which stops the iterative loop on the other threads. The average number of iterations completed on the threads is reported. This number is compared with the case with no allreduce thread (the higher the better).

Figure 8 (right) shows the results on the Linux cluster. MPICH2 demonstrates a better ability than Open MPI to hide the latency of the allreduce.

4 Concluding Remarks

As MPI implementations supporting `MPI_THREAD_MULTIPLE` become increasingly available, there is a need for tests that can shed light on the performance and overhead associated with using multiple threads. We have developed such a test suite and presented its performance on multiple platforms and implementations. The results indicate relatively good performance with MPICH2 and Open MPI on Linux clusters, but poor performance with IBM and Sun MPI on IBM and Sun SMP systems.

We plan to add more tests to the suite, such as to measure the overlap of computation/communication with the MPI-2 file I/O and connect-accept features. We will also accept contributions from others to the test suite. The test suite will be available for download from `www.mcs.anl.gov/~thakur/thread-tests`.

Acknowledgments

We thank the RWTH Aachen University and the San Diego Supercomputer Center for providing computing time on their systems. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

1. García, F., Calderón, A., Carretero, J.: MiMPI: A multithread-safe implementation of MPI. In: Margalef, T., Dongarra, J.J., Luque, E. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1697, pp. 207–214. Springer, Heidelberg (1999)
2. Gropp, W., Thakur, R.: Issues in developing a thread-safe MPI implementation. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)
3. Intel MPI benchmarks, `http://www.intel.com`
4. OSU MPI benchmarks, `http://mvapich.cse.ohio-state.edu/benchmarks`
5. Protopopov, B.V., Skjellum, A.: A multithreaded message passing interface (MPI) architecture: Performance and program issues. Journal of Parallel and Distributed Computing 61(4), 449–466 (2001)
6. Reussner, R., Sanders, P., Träff, J.L.: SKaMPI: A comprehensive benchmark for public benchmarking of MPI. Scientific Programming 10(1), 55–65 (2002)

An Improved Parallel XSL-FO Rendering for Personalized Documents

Luiz Gustavo Fernandes[1], Thiago Nunes[1], Mateus Raeder[1], Fabio Giannetti[2], Alexis Cabeda[3], and Guilherme Bedin[3]

[1] PPGCC/PUCRS - Porto Alegre, Brazil
{gustavo,tnunes,mraeder}@inf.pucrs.br
[2] HP Laboratories - Bristol, United Kingdom
fabio.giannetti@hp.com
[3] HP Brazil R&D - Porto Alegre, Brazil
{alexis.cabeda,guilherme.bedin}@hp.com

Abstract. The use of personalized documents has become a helpful practice on the digital printing area. Automatic procedures to create and transform these documents have become necessary to deal with the growing market demand. Languages such as XSL-FO (eXtensible Stylesheet Language-Formatting Objects) and PPML (Personalized Print Markup Language) have been developed to facilitate the way variable content is inserted within a document. However, these languages have brought together an increasing computational cost of those documents rendering process. Considering that printing shops need to render jobs with thousands of personalized documents within a short period of time, high performance techniques appear as an interesting alternative to improve this rendering process throughput. In this work, we present improvements and new results of a MPI solution previously developed for the FOP (Formatting Objects Processor) tool. FOP is the Apache project rendering tool for personalized documents and its parallel version was optimized in order to allow the computation in parallel of larger input jobs composed of thousands of personalized documents.

1 Introduction

Due to the growing demand for personalized documents, printing high data volume is becoming a common practice among print shops. Most of the digital publishing production environments use digital presses in parallel to maximize the overall document production (jobs). In such an environment, all activities related to the document preparation need to be executed in a constrained time slot, since jobs are completed in a sequential order. The main step on the document preparation is the rendering phase, which is usually quite expensive and in case of thousands of documents it easily becomes the bottleneck of the whole process. Nowadays, modern digital presses are used in parallel, increasing the overall printing speed significantly. In this context, it can become very difficult to feed all the presses with the necessary rendered documents throughput to make the most of their printing speed.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 56–63, 2007.

Similarly to the concept of using presses in parallel to achieve better performance and quickly consume jobs, researches have been carried out to propose a parallelization of the rendering engines. Previous results [1] show that the proposed solution can match the presses speed at the rendering stage, but only for jobs with at maximum 2,000 documents (which have the same structure of test cases described at table 1). The aim of the present work is to optimize this previous parallel solution in such a way it could be used to larger input jobs. A final constraint must be observed: the proposed solution must be restricted to execute over a small set of processors. That is necessary once print shops (which are the target users) do not intend to invest on very expensive and complex parallel platforms.

The remainder of this paper is organized as follows: Section 2 presents in details the research context of rendering personalized documents. Section 3 describes the improvements added to the previous parallel solution. An analysis for experimental results of a test cases set is reported on Section 4. Finally, some concluding remarks and future works are discussed in Section 5.

2 Rendering Personalized Documents

The combination of PPML (Personalized Print Markup Language) [2] and XSL-FO (eXtensible Stylesheet Language-Formatting Objects) [3] has been employed to represent document templates with a high degree of flexibility, reusability and printing optimization. The main advantage of this combination is the expressibility of invariant portions of the template as re-usable objects and variable parts as XSL-FO fragments. Once the variable data is merged into the template, various document instances are formed. The final step is to compose or render the XSL-FO parts into a Page Description Language (PDL). Before describing the processing environment, it is relevant to highlight the most important aspects of these XML (eXtensible Markup Language) [4] formats: PPML and XSL-FO.

PPML is a standard language for digital print built from XML developed by PODi (Print on Demand Initiative) [5] which is a consortium of leading companies in digital printing. PPML has been designed to improve the rendering process of the content of documents using traditional printing languages. It is a hierarchical language that contains documents, pages and objects, which are classified as reusable or disposable. The reusable content represents the concept where data, which is used on many pages, can be sent to the printer once and accessed as many times as needed from the printer's memory. This concept allows high graphical content to be rendered once as well and to be accessed by sending layout instructions instead of re-sending the whole graphic every time it is printed [6].

The variable data is merged and formatted inside the PPML object element using XSL-FO. The containing PPML element is named "copy-hole", which is a defined area in the PPML code that can contain the variable data expressed in XSL-FO, or a non variable content such as images. XSL-FO (also abbreviated as FO - from Formatting Objects) is a W3C [7] standard introduced to format

XML content for paginated media. It ideally works in conjunction with XSL-T (eXtensible Styleshee Language - Transformations) [8] to map XML content into formatted content. Once the XSL-FO is completed with the formatted content, its rendering engine executes the composition step to lay out the content inside the pages and obtain the final composed document. The rendering process in our environment is carried out by Apache's rendering engine named FOP (Formatting Objects Processor) [9].

FOP is one of the most common processors in the market not only because it is an open source application, but also because it provides a variety of output formats and it is flexible and easily extendable. It is a Java application that reads formatting objects and renders to different output formats such as PDF (Portable Document Format), plain text, Post Script, SVG (Scalable Vector Graphics), among others. The rendering process is typically composed of three distinctive steps:

1. Generation of a formatting object tree and properties resolution;
2. Generation of an area tree representing the document having as leafs text elements or images;
3. Converting or mapping the area tree to the output format.

The advantages of this approach are related to the complete independence between the XSL-FO representation of the document and the internal area tree construction. Using this approach makes it possible to map the area tree to a different set of PDLs.

3 Improving Parallel FOP

The parallel implementation for Apache's rendering tool (FOP) discussed on this section was first proposed in [1]. It was designed aiming to speed up the performance of the sequential tool increasing its capacity to feed properly large presses with rendered documents. The basic restriction of this research was the necessity to avoid the use of parallel programming models oriented to very expensive (but not frequently used) platforms because the target users (print shops) do not want to invest on expensive hardware. Typically, useful parallel versions for this tool should run distributed over a small number of processors connected by a fast network. Therefore, the natural choice was a cluster with a message passing programming model. Considering that the original FOP tool was developed in Java for portability reasons, its parallel version should keep this feature matching Java with MPI (Message Passing Interface) [10]. This was achieved through the mpiJava [11] implementation.

After the analysis of the original work, two possible optimization points were detected: (i) **the distribution of data among the rendering modules**, and (ii) **the generation of the final document**. These two factors acting together have limited the scalability of the proposed parallel version, not allowing performance benefits in the rendering process of input jobs with more than 2,000 documents. At this point, it is important to mention that there is a multitude of

possible configurations of documents that can be represented matching PPML and XSL-FO. Therefore, in order to provide a standard to allow comparisons, documents in the context of this work have a set of characteristics that will be presented to readers in the next section.

A detailed description of the original parallel FOP tool will not be discussed here. Readers interested in more details should read [1]. The improvements proposed in this paper will be discussed during the explanation of the new parallel architecture of the FOP tool showed in Fig. 1.

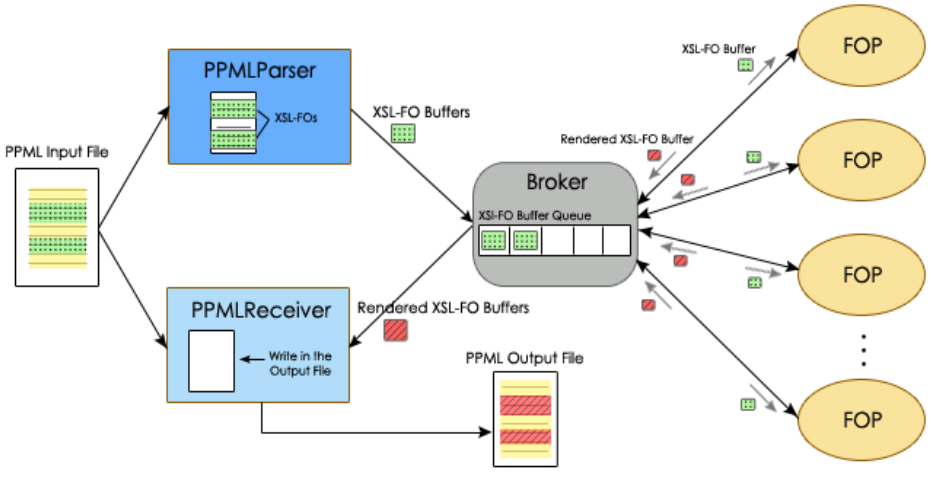


Fig. 1. New parallel FOP architecture

The improved parallel architecture is composed of four modules as follows: PPMLParser, PPMLReceiver, Broker and FOP. The PPMLParser is responsible for parsing the input PPML document, separating variable content to be rendered (XSL-FO) from invariant parts of the document. Considering that the parallel FOP tool is supposed to deal with jobs containing thousands of documents, the parse procedure takes thus a long time. Therefore, it is interesting to provide a way for the rendering modules (called FOPs) to carry out the XSL-FO formatting in parallel with the parsing. This is possible due to the module called Broker. The Broker implements a queue, which contains buffers fulfilled with XSL-FOs found at the input documents, and works in a producer/consumer model using threads. Buffers are generated by the PPMLParser and consumed by the FOPs modules.

The **first optimization** is related to the distribution of the communication buffers. In the first parallel version of the FOP tool, the Broker started to distribute buffers with XSL-FOs to FOPs modules as soon as it received the first buffer from the PPMLParser. Besides, the buffers had a fixed size, not taking into account the number of FOP modules running. In the new parallel version, this procedure was optimized by two modifications: (i) the Broker now waits for having at least one buffer per FOP module queued to start the distribution, and

(ii) the size of the buffers is now calculated considering the amount of FOP modules running. The first optimization avoids the Broker receiving new requests for more work from a FOP module before all FOP modules have work to do. The second modification introduces bigger buffers as the number of available FOP modules grows, which allows the Broker to deal with all FOP modules without making them wait due to concurrent messages. Both modifications help to improve the Broker scalability, increasing the lack of time between the reception of buffers with rendered content from the FOP modules.

FOP modules remain idle until they receive a XSL-FO buffer and start to render its content. When the render is completed, each FOP module sends back to the Broker the result indicating that it is now free to receive more work. FOP modules could send results directly to the PPML Receiver, but this operation would double the number of messages sent by each FOP module: one for the Broker, indicating that it has completed its work and another for the PPMLReceiver with the results. Obviously, this is not interesting because FOP modules should remain rendering as long as possible.

Once the Broker receives rendered buffers from FOP modules, it then sends the results to the PPMLReceiver. Before, this operation along with the XSL-FOs buffers management created an overhead for the Broker. Now, the Broker can easily deal with all these operations without being the bottleneck of the system, because it is not interrupted frequently to manage the XSL-FO buffers.

In the original parallel version, the PPMLReceiver put the rendered FOs in the output file each time a new buffer with results arrived. However, with a larger number of FOP modules running, the PPMLReceiver was overloaded. This fact did not affect the FOP modules, because the communication with the PPMLReceiver was asynchronous. Nevertheless, at the end of all rendering process, a large queue of rendered FOs was formed in the PPMLReceiver waiting to be inserted into the output file. The **second optimization** improves this procedure. In fact, now the PPMLParser gives an identification for each XSL-FO extracted from the input file. The identification helps the PPMLReceiver to avoid a sequential seek in the output file. Plus, while waiting for results and mainly in parallel with the Broker initialization, the PPMLReceiver reads and parses a copy of the input file, storing all the static content in a temporary structure in memory. Each time a buffer with rendered FOs arrives, the operation now consists of locating the right position of the identified FO in memory. As the temporary structure is filled, it is flushed to the output file in parts. This simple procedure saves time and does not require any special component for performing I/O operations in parallel.

4 Experimental Results

The experiments performed to test the improvements proposed in this paper have been carried out in the same environment using the same input data according to the following description. An analysis of the obtained results is done at the end of this section.

4.1 Platform

The software platform adopted to implement the solution proposed is composed of the Java Standard Development Kit (J2SDK, version 1.4.2) plus the standard Message Passing Interface (MPI) [10] to provide communication among processes. More specifically, we choose the **mpich** implementation (version 1.2.6) along with mpiJava [11] (version 1.2.5) which is an object-oriented Java interface to the standard MPI. The experiments were carried out over processors running Linux. The target hardware platform is a multi-computer composed of 14 AMD Opteron 2.4 Ghz nodes, each one with 8GB RAM and connected through a 1 Gb network.

4.2 Input Data

PPML files may appear in a large variety of layouts presenting different number of pages, XSL-FOs and amount of content to be rendered. The size in bytes of a PPML file content is directly related to the computational cost of its rendering. The test cases presented in this section have their layouts as close as possible to the input files used in [1]. However, all test cases used in this work are composed of 20,000 documents. The amount of documents was substantially increased in order to take advantage of the higher computational power offered by the hardware platform aiming to stress the scalability of the improved parallel FOP. The characteristics of these input files are summarized in Tab. 1.

Table 1. Summary of the test cases

File Label	Pages	XSL-FOs	Size (Mb)	Sequential Time (s)
Earth	2	6	160	2,800.77
Fire	2	4	105	2,283.25
Wind	2	8	273	6,285.98

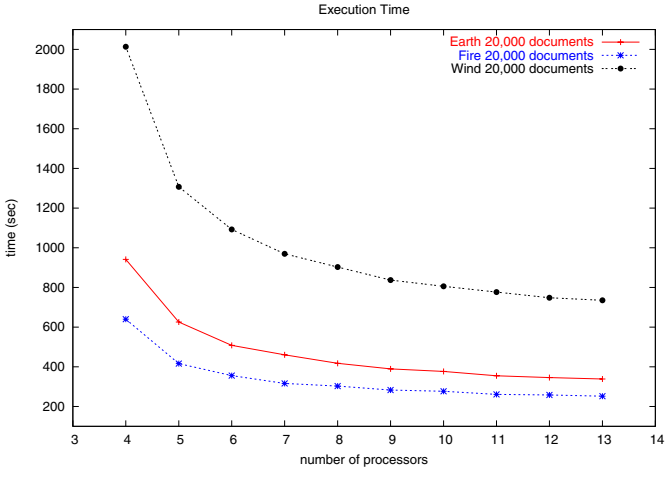
The first input PPML file (**Fire**) is the smallest test case. The amount of the variable content to be rendered is around 105 MB and each of its documents has 2 pages: the first page has only one XSL-FO and the second has 3 XSL-FOs. The second test case (**Earth**) is also composed of 2 pages, each one of them with 3 XSL-FOs. This file has 160 MB of variable content to be rendered. Finally, the last input file (**Wind**) also has 2 pages configured as follows: the first one has 3 XSL-FOs and the second one 5 XSL-FOs. It is the largest PPML file, having 273 MB of variable content.

4.3 Results Analysis

The results presented in this section were obtained through a mean of 10 executions discarding the lowest and the highest times. This procedure is required in order to minimize the influence of environment fluctuations in the final results.

Comparing the sequential times in Tab. 1 with the results in Fig. 2, one can observe the performance gains achieved by the improved parallel FOP. It

is important to notice that the parallel FOP needs at least 4 processors to be executed, one for each module described in Sec. 3. That is the reason why there are no execution times for configurations with 2 and 3 processors. The execution with 5 processors means that there were 2 FOP modules running, 6 processors means 3 FOP modules and so on. As expected, the 4 processors execution has a better performance than the sequential version, even though it has just one process rendering. This happens because there are now time intersections between the parse of the input file, the rendering process and the writing of the output file.



		number of processors									
		4	5	6	7	8	9	10	11	12	13
Earth	T	941.51	625.56	507.99	460.46	417.81	389.51	376.61	354.58	345.49	338.73
	S	2.98	4.78	5.51	6.08	6.70	7.19	7.44	7.89	8.11	8.27
Fire	T	639.98	476.02	385.88	336.37	302.66	282.90	276.43	260.75	258.26	251.95
	S	3.57	4.79	5.92	6.78	7.54	8.07	8.26	8.75	8.84	9.06
Wind	T	2,013.39	1,307.33	1,092.30	969.71	902.99	836.94	805.70	776.85	748.33	735.11
	S	3.12	4.81	5.75	6.48	6.96	7.51	7.80	8.09	8.40	8.55

Fig. 2. Results: execution time (T, in seconds) and speedup (S)

5 Conclusions

The results presented in this work indicate that it is possible to achieve promising results rendering XSL-FO personalized documents using the message passing paradigm over a multi-computer. In this improved parallel version of the one introduced by the authors in a previous work [1], the FOP tool was optimized to allow the rendering of larger input jobs (20,000 documents). In the largest input case, it was possible to decrease the execution time from 6,285.98 to 735.11 seconds over 13 processors without using a prohibitive cost hardware platform.

Despite the results obtained so far, it is the authors' opinion that there are still others improvements which can be added to the parallel FOP in order to achieve higher performance gains. One potential research line is to forward the analysis of the PPML template content in order to previously identify characteristics of the input documents. In this scenario, a smarter version of the parallel FOP would be capable to take advantage of this information to automatically configure itself aiming the best possible performance.

References

1. Giannetti, F., Fernandes, L.G., Timmers, R., Nunes, T., Raeder, M., Castro, M.: High performance XSL-FO rendering for variable data printing. In: ACM Symposium on Applied Computing, Dijon, France, pp. 811–817. ACM Press, New York (2006)
2. Davis, P., de Bronkart, D.: PPML (Personalized Print Markup Language). In: Proceedings of the XML Europe 2000, Paris, France, International Digital Enterprise Alliance (2000)
3. XSL-FO: The Extensible Stylesheet Language Family. section XSL Formatting Objects April 26th, 2007 Extracted from http://www.w3.org/Style/{X}{S}{L}
4. XML: Extensible Markup Language April 26th, 2007 Extracted from http://www.w3.org/{X}{M}{L}
5. PODi: Print on Demand Initiative April 26th, 2007 Extracted from http://www.podi.org/
6. Bosschere, D.D.: Book ticket files & imposition templates for variable data printing fundamentals for PPML. In: Proceedings of the XML Europe 2000, Paris, France, International Digital Enterprise Alliance (2000)
7. W3C: The World Wide Web Consortium April 26th, 2007 Extracted from http://www.w3.org/
8. XSL-T: XSL-Transformations section References April 26th, 2007 Extracted from http://www.w3.org/TR/1999/REC-xslt-19991116
9. FOP: Formatting Objects Processor April 26th, 2007 Extracted from http://xml.apache.org/fop/
10. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: the complete reference. MIT Press, Cambridge (1996)
11. mpiJava: The mpiJava Home Page April 26th, 2007 Extracted from http://www.hpjava.org/mpiJava.html

An Extensible Framework for Distributed Testing of MPI Implementations

Joshua Hursey[1], Ethan Mallove[2], Jeffrey M. Squyres[3], and Andrew Lumsdaine[1]

[1] Indiana University Open Systems Laboratory Bloomington, IN USA
{jjhursey,lums}@osl.iu.edu
[2] Sun Microsystems, Inc. Burlington, MA USA
ethan.mallove@sun.com
[3] Cisco, Inc. San Jose, CA USA
jsquyres@cisco.com

Abstract. Complex code bases require continual testing to ensure that both new development and routine maintenance do not create unintended side effects. Automation of regression testing is a common mechanism to ensure consistency, accuracy, and repeatability of results. The MPI Testing Tool (MTT) is a flexible framework specifically designed for testing MPI implementations across multiple organizations and environments. The MTT offers a unique combination of features not available in any individual testing framework, including a built-in multiplicative effect for creating and running tests, historical correctness and performance analysis, and support for multiple cluster resource managers.

1 Introduction

High quality MPI implementations are software packages so large and complex that automated testing is *required* to effectively develop and maintain them. Performance is just as important as correctness in MPI implementations, and therefore must be an integral part of the regression testing assessment. However, the number of individual tests taken in combination with portability requirements, scalability needs, and runtime parameters generates an enormous set of testing dimensions. The resulting testing space is so large that no single organization can fully test an MPI implementation. Therefore, a testing framework suitable for MPI implementations must be able to combine testing results from multiple organizations to generate a complete view of the testing coverage.

Many MPI test suites and benchmarks already exist that can verify the correctness and performance of an MPI implementation. Additionally, MPI implementation projects tend to have their own internal collection of tests. However, running a large set of tests manually on a regular basis is problematic; human error and changing underlying environments will cause repeatability issues.

A good method for regression testing in large software projects is to incorporate automated testing and reporting, run on a regular basis. Abstractly, a testing framework is required to: obtain and build the software to test; obtain

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 64–72, 2007.

and build individual tests; run all tests variations; and report both detailed and aggregated testing results. Additionally, since the High Performance Computing (HPC) community produces open source implementations of MPI that include contributions from many different organizations, MPI implementation testing methodology and technology must also:

- Be freely available to minimize the deployment cost.
- Easily incorporate thousands of existing MPI tests.
- Support simultaneous distributed testing across multiple sites, including operating behind organizational security boundaries (e.g., firewalls).
- Support on-demand reporting, specialization, and email reports.
- Support execution of parallel tests, and therefore also support a variety of cluster resource managers.

We have therefore created the MPI Testing Tool (MTT), an MPI implementation-agnostic testing tool to satisfy these needs, and have prototyped its use in the Open MPI project [1].

The rest of this paper is organized as follows: related work is presented in Section 2. Section 3 describes the MPI Testing Tool (MTT) in more detail. Section 4 presents MTT experiences with the Open MPI project. Finally, we present conclusions and a selection of future work items in Section 5.

2 Related Work

There is a large field of research surrounding optimal testing techniques, but only a few of those ideas seem to have any impact on the software engineering process used to develop and maintain large software systems [2,3]. To maintain the high quality of large software systems, continual regression testing is required. Onoma et al. [4] describe vital components of an effective software testing suite. The MTT is designed to satisfy these requirements for MPI implementations.

TET [5] is a widely adopted tool among hundreds of free and open source testing frameworks. However, TET does not provide mechanisms for obtaining the software to be tested, therefore requiring an additional layer of software to determine whether new versions are available to prevent duplicate testing results. TET also has a crude reporting mechanism which requires searching through flat file logs for test results. Although logs can be exported to a database for historical data mining, no front-end is provided for querying the test results.

Perfbase [6] presents a front-end to a SQL database for storing historical performance data. Perfbase does not provide a framework for obtaining, building, and testing software, but rather focuses on archiving and querying test results. Although the first generation of MTT used Perfbase as a back-end data store, MTT evolved its own data store mechanisms due to inflexibility of Perfbase's storage and retrieval model.

DejaGNU [7] is a testing framework from which testing harnesses and suites can be derived. Similar to TET, no rich reporting mechanisms are provided, and native support for parallelism is not included. DejaGNU also requires individualized test suite scripts for each test conducted.

Many other testing harnesses are available, including Kitware's Dart system, the Boost C++ regression testing system, Mozilla Tinderbox and Testopia, and the buildbot project. However, none of the products and projects surveyed met the full set of requirements needed for testing MPI implementations in a distributed, scalable fashion.

3 The MPI Testing Tool (MTT)

The MTT was created to solve many of the issues cited above.

At its core, the MTT is a testing engine that executes the following phases:

1. **MPI Get**: Obtain MPI implementations. Implementations are obtained from the Internet (via HTTP/S, FTP, Subversion), local copies (e.g., tarball or directory), or by reference to a working installation.
2. **MPI Install**: Specify how to build/install MPI implementations.
3. **Test Get**: Obtain test suite source codes, similar to the MPI Get phase.
4. **Test Build**: Compile test suites against all successfully installed MPI implementations.
5. **Test Run**: Run individual tests in each successfully built test suite.

The MPI Install, Test Build, and Test Run phase results are stored in a central database (or other user specified mechanism). All MTT tests will generate one of four possible results: pass, fail, timeout, or skip. Each test defines specific criteria indicating success. For example, an MPI implementation must compile and install successfully to qualify as "pass." A test is "failed" if the test completes but the "pass" criteria is not met. A test is killed and declared a "timeout" if its execution did not finish within the allotted time period. A test is declared "skip" if it elects not to run. For example, an InfiniBand-specific test may opt to be skipped if there are no InfiniBand networks available.

Phases are templates which allow multiple executions of each step based on parametrization. Later phases are combined with all successful invocations of prior phase invocations, creating a natural multiplicative effect.

- M: MPI implementations
- I: MPI installations, each of which are applied against all M MPI implementations.
- N_t: Individual tests, grouped by test suite (t), each of which is compiled against all ($M \times I$) MPI installations.
- R_t: Run parameters specified for tests, each of which is applied against ($M \times I \times N_t$) tests.

Fig. 1 shows the general sequence of the phases as well as their relationships to each other and the natural multiplicative effect. The figure shows one MPI implementation that is installed two different ways (e.g., with two different compiler suites). Each installation is then used to build two tests; each test is run

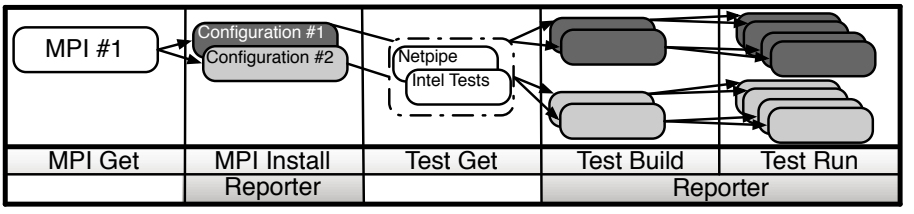


Fig. 1. MTT phase relationship diagram. Phases labeled with "Reporter" contain tests that are saved to a back-end data store such as a database.

two different ways. Hence, the total number of reported tests will follow the equation:

$$num\_MPI\_installs + num\_test\_builds + num\_test\_runs$$
$$(M \times I) + \quad (M \times I \times N_t) \quad + (M \times I \times N_t \times R_t)$$

Therefore Fig. 1 shows a total of 14 tests (2 MPI installs + 4 test builds + 8 test runs). While the multiplicative effect is a deliberate design decision, users must be careful to not create test sets that incur prohibitively long run times.

3.1 Configuration

The MTT test engine is configured by an INI-style text file specified on the command line. In the INI file format, sections are denoted with strings inside brackets and parameters are specified as `key = value` pairs. Fig. 2 shows a sample fragment of an MTT INI configuration file.

A typical configuration file contains a global parameters section, one or more MPI Details sections, and one or more sections for each of the execution phases.

The global parameters section is used to specify testing parameters and user preferences across an entire run of the MTT. MPI Details sections specify how to run executables for a specific MPI. For example, these sections contain nuances such as whether `mpirun` or `mpiexec` should be used, what command line options to use, etc. Each execution phase will also be specified by at least one section in the configuration file. The phase INI sections are comprised of phase-specific parameters, the designation of a plugin module to use, and module-specific parameters. Fig. 2 shows an INI file example that downloads two different versions of Open MPI detailed in the MPI Get phases and compiles both of them with two different compilers (GNU, Intel) detailed in the two MPI Install phases.

Any number of MPI implementations can be specified for download in MPI Get sections. Since each MPI Get section will potentially download a different MPI implementation (and therefore require a different installation process), MPI Install sections must specify which MPI Get section(s) to install.

Phases are linked together in the configuration file by back-referencing one or more prior phase names. For example, lines 10 and 14 in Fig. 2 show the two MPI Install sections back-referencing the "Open MPI nightly trunk" and "Open

```
1  [MPI Get: Open MPI nightly trunk]
2  module = SVN
3  svn_url = http://svn.open-mpi.org/svn/ompi/trunk
4
5  [MPI Get: Open MPI v1.2 snapshots]
6  module = OMPI_Snapshot
7  ompi_snapshot_url = http://www.open-mpi.org/nightly/v1.2
8
9  [MPI Install: GNU compilers]
10 mpi_get = Open MPI nightly trunk,Open MPI v1.2 snapshots
11 module = OMPI
12
13 [MPI Install: Intel compilers]
14 mpi_get = Open MPI nightly trunk,Open MPI v1.2 snapshots
15 module = OMPI
16 ompi_configure_arguments = CC=icc CXX=icpc F77=ifort FC=ifort CFLAGS=-g
17
18 [MPI Details: Open MPI]
19 exec = mpirun --mca btl self,&enumerate(``tcp'', ``openib'') \
20     -np &test_np() &test_executable()
```

Fig. 2. Fragment of a simplified MTT configuration file. Two MPI Get phases are paired with two MPI Install phases, resulting in four MPI installations. The MPI EAM: template is not listed as a verb on dictionary.com, but oh well :) Details section templates an execution command for invoking MPI tests. This example shows at least two executions for each test: one each for TCP and OpenFabrics networks.

MPI v1.2 snapshots" MPI Get sections. Hence, these two MPI Install sections will be used to install both MPI Get sections. Similar back-referencing mechanisms are used for the other phases.

MTT will conditionally execute phases based on the outcome of prior phases. For example, the MPI Install phase is only executed in the case where the prior execution of the corresponding MPI Get phase was both successful and yielded a *new* version of the MPI implementation (unless otherwise specified). Similarly, Test Run sections will only execute tests where all prior phases were successful: a new MPI implementation was obtained and successfully installed, and tests were successfully obtained and compiled against the MPI installation.

3.2 Funclets

Additional combinations of testing parameters can be specified via "funclets" in the configuration file. "Funclets" are Perl-like function invocations that provide both conditional logic and and text expansion capabilities in the configuration file, enabling it to serve as a template that is applicable to a variety of different scenarios.

A common use of funclets is to expand a configuration parameter to be an array of values. For example, the `np` parameter in Test Run sections specifies how many processes to run in the test. `np` can be set to one or more integer values.

The following example assigns an array of values to the `np` parameter by using three funclets:

```
np = &pow(2, 0, &log(2, &env_max_procs()))
```

- `&env_max_procs()`: Returns the maximum number of processes allowed in this environment (e.g., number of processors available in a SLURM or Torque job, the number of hosts in a hostfile, etc.).
- `&log()`: Returns the log of the first parameter to the second parameter.
- `&pow()`: Returns an array of integer values. The first parameter is the base, the second and third parameters are the minimum and maximum exponents, respectively.

In the above example, when running in a SLURM job of 32 processors, `np` would be assigned an array containing the values 2, 4, 8, 16, and 32. This causes the MTT execution engine to run each test multiple times: one for each value in the array.

3.3 Test Specification

The MTT supports adding tests in a modular and extensible manner. The procedures to obtain and build test suites (or individual tests) are specified in the configuration file. Although the MTT can execute arbitrary shell commands from the configuration file to build test suites, complex build scenarios are typically better performed via MTT plugin modules. Tests to run are also specified in the configuration file; the MTT provides fine-grained control over grouping of tests, pass/fail conditions, timeout values, and other run-time attributes.

3.4 Test Execution

The MPI Details section tells the MTT how to run an executable with a particular MPI implementation. Each MPI Get section forward-references an MPI Details section that describes how to run executables for that MPI. This feature allows the MTT to be MPI-implementation-agnostic.

For example, line 18 inFig. 2 shows an MPI Details section for Open MPI. The `exec` parameter provides a command line template to run tests. The funclets `&test_np()` and `&test_executable()` are available to "paste in" the values specific to the individual test being invoked. Note, too, the use of the `&enumerate()` funclet. This funclet will return an array of all of the values passed as parameters, effectively causing the `exec` parameter to expand into at least *two* `mpirun` command lines: one with the string "`--mca btl self,tcp`" and another with the string "`--mca btl self,openib`" (forcing Open MPI to use the TCP and OpenFabrics network transports, respectively).

Combining the use of multiple funclets can result in a multiplicative effect. For example, using the same 32 processor SLURM job and funclet-driven `np` value from Section 3.2, the `exec` parameter from Fig. 2 will expand to invoke *ten* command lines for each test: five with the TCP transport (with 2, 4, 8, 16, and 32 processes), and five with the OpenFabrics transport.

3.5 Reporting Testing Results

Fig. 1 illustrates that the Reporter phase is run after the MPI Install, Test Build, and Test Run phases. The Reporter phase writes testing results to a back-end data store such as a central database, but may also log information to local text files (or a terminal).

The MTT features a web interface to the central database to facilitate complex interactive explorations of the testing data, including comparisons of performance and correctness over multiple versions of an MPI implementation. The web interface is specifically designed to aggregate the testing results into high level summary reports that can be used to repetitively narrow searches in order to find specific data points. Such "drill down" methods are commonly used to aid in discovering trends in test failures, displaying historical performance results, comparing results between different configurations, etc.

Additionally the web-based reporter interface provides custom stored queries, allowing developers to easily share views of the testing data. Absolute and relative date range reports are useful when citing a specific testing result and tracking its progress over time, respectively.

4 Case Study: The Open MPI Project

The Open MPI project relies on the MTT for daily correctness and performance regression testing. Open MPI is a portable implementation of the MPI standard that can run in a wide variety of environments; testing it entails traversing a complex parameter space due to the enormous number of possible environments, networks, configurations, and run-time tunable parameters supported.

The enormous parameter space, in combination with limited available testing resources, prevents any one member organization from achieving complete code coverage in their testing. By running the MTT at each Open MPI member organization, the project effectively pools the resources of all members and is able to achieve an adequate level of testing code coverage. This scheme naturally allows each organization to test only the specific configurations that are important to their goals.

Member testing resources range from small to large collections of machines; unscheduled and scheduled environments; with and without firewall restrictions. Each organization has their own site-specific MTT configuration files that detail exactly which scenarios and environments to test.

The Open MPI project has two distinct testing schedules (weekday for "short" 24-hour testing, and weekend for longer / higher process count testing); both follow the same general format: Open MPI snapshot tarballs are generated from the Open MPI Subversion development trunk and release branches and are posted on the Open MPI website. Member organizations use the MTT to download and test the snapshots on their local testing resources.

Several well-known MPI test suites are run against Open MPI via the MTT, including the Intel MPI test suite, the LAM/IBM MPI test suite, the Notre

Dame C++ MPI test suite, the Intel MPI Benchmarks (IMB), NetPIPE, etc. Many tests internal to the Open MPI project are also run via the MTT.

As each member's testing completes, results are uploaded to a central database hosted by Indiana University and made available through the MTT's web-based reporter interface. On weekdays, rollup summary reports are e-mailed to the Open MPI development team 12 and 24 hours after the daily cycle begins. Summary reports of weekend-long testing are sent on Monday morning. The e-mail reports, combined with detailed drill-down queries, form the basis of daily discussions among developers, corrections and modifications to recent changes, and decisions about release schedules.

Using the MTT, the Open MPI project has accumulated over 7 million test results between November 2006 and May 2007. Approximately 100,000 tests are run each weekday/weekend cycle, spanning six platforms, six compiler suites, and seven network transports.

5 Conclusions

The MTT successfully supports the active development of the Open MPI project providing correctness and performance regression testing. The MTT provides a full suite of functionality useful in the automated routine testing required of a high quality MPI implementation. By fully automating the testing process, developers spend more time developing software than routinely testing it. Although this paper has focused on the MTT's testing of Open MPI, the MTT is MPI implementation agnostic, and has been used to test other MPI implementations, such as LAM/MPI [8] and MPICH2 [9]. The MTT is available at:
`http://www.open-mpi.org/projects/mtt/`

5.1 Future Work

While the MTT currently supports many modes of operation, there are several areas where its support could be expanded, including: supporting testing in heterogeneous environments, supporting complex "disconnected" scenarios for environments not directly (or even indirectly) connected to the Internet, exploiting the natural parallelism exhibited by orthogonal steps within the MTT testing cycle to make more efficient use of testing resources, and expanding the MTT to support general middleware testing.

References

1. Gabriel, E., et al.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104 (2004)
2. Osterweil, L.: Strategic directions in software quality. ACM Comput. Surv. 28(4), 738–750 (1996)
3. Harrold, M.J.: Testing: a roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, pp. 61–72. ACM Press, New York (2000)

4. Onoma, A.K., Tsai, W.T., Poonawala, M., Suganuma, H.: Regression testing in an industrial environment. Commun. ACM 41(5), 81–86 (1998)
5. TET Team: TETware white paper. Technical report, The Open Group (2005), http://tetworks.opengroup.org/Wpapers/TETwareWhitePaper.htm
6. Worringen, J.: Experiment management and analysis with perfbase. In: IEEE Cluster Computing 2005, pp. 1–11. IEEE Computer Society Press, Los Alamitos (2005)
7. Free Software Foundation: DejaGnu (2006), http://www.gnu.org/software/dejagnu/
8. Squyres, J.M., Lumsdaine, A.: A Component Architecture for LAM/MPI. In: Dongarra, J.J., Laforenza, D., Orlando, S. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2840, pp. 379–387. Springer, Heidelberg (2003)
9. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22(6), 789–828 (1996)

A Virtual Test Environment for MPI Development: Quick Answers to Many Small Questions

Wolfgang Schnerring, Christian Kauhaus, and Dietmar Fey

Lehrstuhl für Rechnerarchitektur und -kommunikation, Institut für Informatik,
Friedrich-Schiller-Universität, 07737 Jena, Germany
{wosc,kauhaus,fey}@cs.uni-jena.de

Abstract. MPI implementations are faced with growingly complex network configurations containing multiple network interfaces per node, NAT, or dual stacks. To implement handling logic correctly, thorough testing is necessary. However, the cost of providing such diverse setups in real hardware is prohibitively high, resulting in a lack of testing. In this article, we present a *Virtual Test Environment (VTE)* that considerably lowers this barrier by providing complex network environments on a single computer and thus enables testing in situations where it otherwise would not be feasible.

1 Introduction

As today's cluster computers become more and more sophisticated, complex network setups are increasingly common. Of course, application writers do not want to be bothered with network topology and expect their MPI implementation to cover the details. In consequence, nearly any modern MPI library contains nontrivial amounts of logic to perform the initial wire-up: starting daemons, querying addresses, selecting network interfaces, etc.

To implement wire-up logic code properly, frequent and thorough testing is necessary. During the development of our IPv6 extension to Open MPI [1], we ran into the problem of verifying correct behaviour on configurations like clusters with multiple networks, multi-domain clusters using both private and public IPv4 addressing, mixed IPv4/IPv6 environments, and others. The cost of providing such setups was prohibitively high, resulting in a lack of testing and, in consequence, undiscovered bugs.

To remedy this problem, we present a *Virtual Test Environment*[1] that lowers the testing effort significantly, so that the execution of some classes of functional tests becomes practicable at all. Additionally, it is lightweight enough to facilitate a rapid feedback cycle during development. VTE builds virtual clusters on the developer's workstation from high-level descriptions by employing kernel virtualisation. The *Unit Under Test (UUT),* e. g., a MPI implementation, is exercised in a variety of network setups. Although VTE was created to facilitate

[1] The software can be obtained from http://www2.informatik.uni-jena.de/cmc/vte/.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 73–80, 2007.

the development of our IPv6 extension to Open MPI, it is not constrained to this task. In fact, VTE is useful for the development of any distributed application which interacts with a TCP/IP network environment.

This paper is organised as follows. Section 2 explains the design and implementation of VTE. Section 3 illustrates the capabilities of VTE using real-world examples. Section 4 concludes and outlines VTE's further development.

2 Design and Implementation

The aim of VTE is to lower the cost associated with the testing process. As VTE has been specifically designed to reflect this goal, we have made several architectural decisions. We first give an overview over those decisions, and then present them in more detail.

First, VTE needs a concise description language for test networks. If it is too difficult to specify the test setup (e. g., spending a day in the machine room installing cables), the developer is very likely to skip testing. Second, tests should run quickly. If tests take longer than a few minutes, it is too tedious to test frequently. Third, the test environment should be transparent to the UUT, requiring no test-specific modifications. But the test environment should *not* be transparent to the developer, providing him with effective control and inspection devices. Fourth, the test environment should be able to model complex network configurations to be representative of most cluster environments seen today.

Related to our work are MLN [2], an administration tool for virtual machines that allows a declarative description, and Xen-OSCAR [3], a virtualisation-based tool to test cluster installation processes. Unfortunately, both are not fit for MPI implementation testing, since they are not optimised for short running time and provide no means to reconfigure the network during runtime.

2.1 Concise Description of Network Configurations

To provide the developer with concise means of specifying tests, we designed a domain-specific language to describe network configurations. The virtual network can be built out of components like *host* and *switch*, which model physical devices, but do more than their real-world counterparts. For example, when connecting VMs with a switch, they are assigned IP addresses automatically.

To enable for such concise expression, VTE is implemented in Ruby, a language that allows to express facts on a very high, abstract level. Configurations are executable Ruby programs that specify all necessary information to create virtual machines and networks. While executing a network description, VTE both builds internal data structures describing the configuration and interacts with the host system through shell commands to realise those structures.

Thus, specifying

```
cluster = Cluster.new(16)
switch = Switch.new("192.168.5.0/24").connect(cluster.vm)
```

configures a virtual cluster with 16 nodes that are connected through a switch and have IP addresses from the subnet 192.168.5.0/24.

This example shows some of the most important language elements. The Cluster class governs a set of virtual machines. With Cluster.new, virtual machines are booted and the Cluster.vm accessor provides references to all running VMs. After startup, VMs do not have a network connection. A virtual network is created using Switch.new, which also causes VTE to create the supporting operating system structures (see 2.4). The Switch.connect method creates network interfaces, allocates IP addresses, and creates the connections. More configuration language elements are documented in the API reference.

We designed the configuration language to use sensible defaults, while allowing exceptions to be specified if desired. If the network above was not given explicitly but rather just as the type (using Switch.new(:ipv4)), a suitable IPv4 address range would be allocated automatically.

2.2 Fast Execution

To run fast, VTE requires a lightweight virtualisation technology, since the main overhead stems from running virtual machines. The "smallest" form of virtualisation is kernel-based virtualisation [4], which merely creates a compartment inside the running kernel to isolate processes of one VM from those of other VMs. We have chosen OpenVZ [5] since it is the most mature of the freely available kernel-based virtualisers for Linux and offers thorough network support.

Kernel-based virtualisation allows for rapid creation of virtual machines, since creating a VM mostly consists of spawning another init process and a few required daemons, for example sshd, reusing the already running kernel. This also means that VMs are not able to use a different operating system than the one which is already present, but since VTE focuses on complex network configurations, heterogeneous operating systems are not a primary concern.

We accelerate the startup process even further by reusing one file system image for all virtual machines: Instead of setting up separate file systems for newly created VMs, only one master file system is mounted repeatedly using copy-on-write semantics. Write operations are diverted to a separate location, while the master file system stays read-only. With this technique, VTE is able to create VMs very quickly: Figure 1 shows the total time required for starting VMs and shutting them down again on an Intel Pentium 4 at 3 GHz with 2 GB RAM running kernel version 2.6.18-ovz028test019. Even 50 VMs take only about a minute, but most tests will not require much more than 10 VMs. Since their startup and shutdown takes only 16 seconds, this presents practically no barrier to frequent testing. OpenVZ is also quite efficient: VTE is capable of simulating a cluster with 300 nodes on the same machine. An MPI ring test then takes about 3 minutes, plus about 8 minutes VM startup and shutdown time.

To test the UUT with several network configurations, it should not be necessary to stop the VMs and start them again in a different setup. Therefore, we implemented all virtual network components in a way they can be completely removed and new ones added while the VMs keep running.

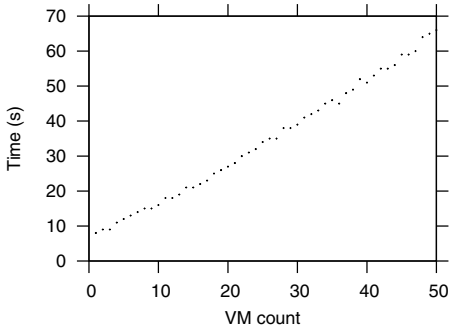


Fig. 1. Time required for startup and shutdown

To bring the UUT into the test environment, it is not necessary to copy it onto all virtual machines. OpenVZ allows to simply mount directories of the host into a VM, which saves a potentially large amount of data transfers. For example, to inject a MPI implementation into the test environment, the developer's working directory can be mounted, even under the same path as on the host, so the MPI implementation does not notice any difference.

2.3 Semi-transparency

VTE is able to run real software, and it appears as a real cluster to the software running inside it. But the developer has full control over this environment, so VTE could be imagined as a semi-transparent mirror.

The test environment can be controlled by executing shell commands inside the virtual machines and examining their output. We implemented running shell commands using OpenVZ's `vzctl exec` mechanism. An alternative would be to inject commands into VMs using `ssh` calls via a virtual service network, which would of course influence the network setup. This way, however, VTE can be configured exactly as required for the tests—even completely without a network. Since the virtual machines export their network interfaces to the host, network sniffers or any other tools can be run on the host to examine traffic on the virtual network.

VTE provides a self-contained environment. Its only external dependencies are OpenVZ to provide the virtualisation, and root privileges for operations concerning the virtual environment, which are encapsulated in `sudo` calls.

2.4 Complex Network Configuration

To be able to test the behaviour of MPI implementations even for complicated cases, VTE should model fairly complex network environments. We limited the network support to TCP/IP though, favouring simplicity and conciseness of the configuration language over heterogeneous network families.

Figure 2 shows an example setup where two clusters are connected via a router. VTE constructs this virtual network on the host, since the network interfaces of

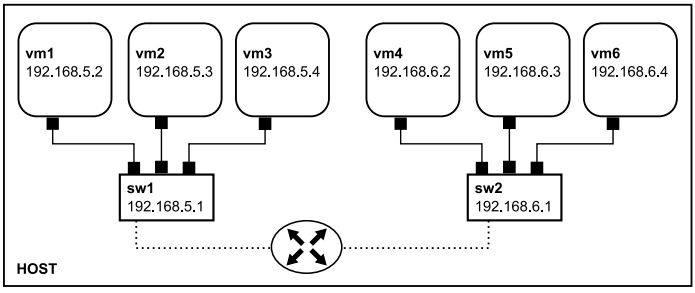


Fig. 2. Example network configuration

the virtual machines are easily accessible from there. Switches are implemented by using standard Linux bridge−utils to create virtual bridges that connect the network interfaces of the VMs. Since the bridges can serve a dual purpose and function as network interfaces of the host, the host itself can also be connected to a virtual switch if desired. We use this to implement a router using the host's routing table. Network Address Translation (NAT) is also performed on the host via iptables. Thus, virtual network components can be combined to model multi-cluster setups, multi-link connections (e. g. channel bonding), dual stack (IPv4/IPv6) networks, and other configurations.

3 Examples

To show the ease of use that VTE offers for specifying and running tests, we examine the behaviour of MPI implementations that occurred to us during our day-to-day development work to show how VTE can help preventing this kind of problems by facilitating regular testing.

3.1 Striping

If there are multiple network paths between any two nodes, Open MPI optionally uses striping to maximise throughput. With striping, Open MPI fragments big messages and sends them in parallel through several interfaces. VTE allows for each VM to have multiple network interfaces, for example by specifying an interface number when connecting VMs with a switch:

```
cluster = Cluster.new(2)
Switch.new(:ipv4).connect(cluster.vm, 0)
Switch.new(:ipv4).connect(cluster.vm, 1)
```

In this example, each VM will be configured with two network interfaces, eth0 and eth1. Running the ping-pong benchmark from the Intel MPI Benchmark Suite (IMB) 3.0 shows that once the payload is large enough, Open MPI will use striping to distribute the traffic over all available interfaces. The tcpdump output shows connections on both switches.

```
# tcpdump -i sw1
IP 1.0.0.2.56003 > 1.0.0.3.33364
IP 1.0.0.3.33364 > 1.0.0.2.56003
# tcpdump -i sw2
IP 2.0.0.2.54115 > 2.0.0.3.59036
IP 2.0.0.3.59036 > 2.0.0.2.54115
```

3.2 NAT

Many clusters use private IPv4 addresses for their nodes, which connect to the outside world using a NAT gateway. Combining two clusters of this kind to a multi-cluster is not possible, since the private addresses of one cluster are not reachable from the other. MPI implementations should detect this erroneous condition. We examine the behaviour of LAM/MPI and Open MPI using VTE.

Two clusters with 3 nodes each that use private IPv4 addresses and are connected with NAT (see Figure 2) can be modelled like this:

```
cluster1 = Cluster.new(3)
cluster2 = Cluster.new(3)
switch1 = Switch.new("192.168.5.0/24").connect(cluster1.vm)
switch2 = Switch.new("192.168.6.0/24").connect(cluster2.vm)
router = Router.new.connect(switch1, nat=true).connect(switch2, nat=true)
```

LAM/MPI [6] is started with the `lamboot` command. The required host file is automatically constructed by concatenating the host names of all VMs.

```
hostnames = (cluster1.vm + cluster2.vm).map { |vm| vm.hostname }.join("\n")
cluster1.vm[0].cmd("echo '#{hostnames}' > lamhosts")
cluster1.vm[0].cmd("lamboot lamhosts")
```

`lamboot` tries to start LAM's management daemon `lamd` on all hosts, but since some hosts use private addresses and are unreachable, it terminates with an error message:

```
ERROR: LAM/MPI unexpectedly received the following on stderr:
ssh: connect to host vm3 port 22: Network is unreachable
```

Open MPI [7] starts its daemon `orted` implicitly when `mpirun` is called.

```
hostnames = (cluster1.vm + cluster2.vm).map { |vm| vm.hostname }.join(",")
cluster1.vm[0].cmd("mpirun -np #{cluster.vm.size} -host #{hostnames} ./ringtest")
```

Open MPI also fails to contact some hosts, but instead of terminating it hangs until VTE's timeout kills it:

```
ssh: connect to host vm3 port 22: Network is unreachable
[vm0:21702] ERROR: A daemon on node vm3 failed to start as expected.
[vm0:21702] ERROR: There may be more information available from
[vm0:21702] ERROR: the remote shell (see above).
Timeout: aborting command "vzctl" with signal 9
```

The problem is that after Open MPI notices the error, it terminates by telling `orted` on all hosts to shut down—including those hosts that it was unable to

connect in the first place. Since these hosts never return a "shutdown successful" notification, the process keeps waiting for them forever.

3.3 Installation Prefix

If Open MPI is installed below a different path than the prefix specified at compile time, it is unable to find its binaries and libraries. One method of explicitly specifying installation locations is to use application context files. The manual page for mpirun states that "[...] --prefix can be set on a per-context basis, allowing for different values for different nodes." With VTE, two nodes with different Open MPI installation locations can be described as follows:

```
cluster = Cluster.new(2)
switch = Switch.new(:ipv4).connect(cluster.vm)
cluster.vm[0].mount("/real/path/to/openmpi", "/usr/local/openmpi")
cluster.vm[1].mount("/real/path/to/openmpi", "/opt/openmpi")
cluster.vm[0].cmd("echo '#{<<_EOT_}' > appfile")
  -np 1 -host vm0 --prefix /usr/local/openmpi hostname
  -np 1 -host vm1 --prefix /opt/openmpi hostname
_EOT_
cluster.vm[0].cmd("mpirun --appfile appfile date")
```

Unfortunately, the --prefix settings does not take effect and mpirun fails with /usr/local/openmpi/bin/orted: No such file or directory on vm_1. Perhaps the writing of the manual page proceeded a little bit quicker than the writing of the code.

3.4 Dual-Stack

When migrating to IPv6 [8], it is quite common to have a dual-stack setup, that is both address families on a single interface, connected by the same switch. In VTE an IP subnet is usually represented by a switch, but in this case there is only one switch but two networks. Therefore we have to configure the second set of IP addresses explicitly, using the provided helper functions:

```
cluster = Cluster.new(3)
Switch.new(:ipv6).connect(cluster.vm)
net = Switch.generate_network(:ipv4)
cluster.vm.each_with_index do |v, i|
  v.configure_ipv4(net.nth(i+1)) unless v.hostname_short == "vm1"
end
Cluster.update_hostfile
```

VTE's configuration language is plain Ruby, so all normal language constructs are available to the developer. This makes it easy to describe exceptions like "every host has IPv4 except vm_1."

Running a ring test and observing the traffic on the switch with tcpdump shows that Open MPI with our IPv6 extension now opens connections via both IPv4 and IPv6, depending on the connectivity available. vm_0 connects to vm_2 via IPv4 (1.0.0.1–1.0.0.3), but to vm_1 via IPv6 (aa01::1–aa01::2).

```
# tcpdump -i sw1
IP 1.0.0.1.33126 > 1.0.0.3.52353
IP 1.0.0.3.52353 > 1.0.0.1.33126
[...]
2001:638:906:aa01::1.38252 > 2001:638:906:aa01::2.43756
2001:638:906:aa01::2.43756 > 2001:638:906:aa01::1.38252
```

These examples show some of the VTE's possibilities to exercise MPI implementations in a variety of environments. The configuration language is expressive enough to formulate even bug-provoking border cases in a compact form.

4 Conclusion

We have presented a Virtual Test Environment that considerably lowers the barrier to functional testing of distributed applications, thereby enabling testing in cases where it were not feasible otherwise, since the costs of physical test setups are too high. VTE has an expressive configuration language for describing complex network environments and short execution times. Being able to run networking tests in just a few minutes allows the developer to establish a close feedback loop, which results in better software quality.

References

1. Kauhaus, C., Knoth, A., Peiselt, T., Fey, D.: Efficient message passing on multiclusters: An IPv6 extension to Open MPI. In: Proceedings of KiCC'07, Chemnitzer Informatik Berichte CSR-07-02 (2007)
2. Begnum, K.M.: Managing large networks of virtual machines. In: Proc. LISA'06: 20th Large Installation System Administration Conference, pp. 205–214. USENIX Association, Washington, D.C (2006)
3. Vallée, G., Scott, S.L.: OSCAR testing with Xen. In: Proc. 20th Int. Symp. on High-Performance Computing in an Advanced Collaborative Environment (HPCS'06), pp. 43–48. IEEE Computer Society, Washington, DC (2006)
4. Soltesz, S., Pötzl, H., Fiuczynski, M., Bavier, A., Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In: Proceedings of EuroSys 2007, Lisbon, Portugal (March 2007)
5. SWSoft: OpenVZ – server virtualization open source project. Accessed on June 27, 2007, http://openvz.org/
6. Squyres, J.M., Lumsdaine, A.: A Component Architecture for LAM/MPI. In: Dongarra, J.J., Laforenza, D., Orlando, S. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2840, pp. 379–387. Springer, Heidelberg (2003)
7. Gabriel, E., Fagg, G.E., Bosilca, G.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104 (2004)
8. Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard) (December 1998)

Multithreaded Tomographic Reconstruction*

José Antonio Álvarez, Javier Roca, and Jose Jesús Fernández

Departamento de Arquitectura de Computadores y Electrónica.
Universidad de Almería
{jaberme,jroca,jose}@ace.ual.es

Abstract. An option to increase performance in grand-challenge applications is overlap computation and communication. In applications developed using C and MPI, this overlapping is usually left to the programmer, but this is a tedious task. Switching from the process model to a threaded model in the parallel environment via user level threads takes advantage of the existing concurrence in an application. In this paper we expose and analyze our research group's tomographic reconstruction software developed using AMPI, a user level thread framework that provides a solution to port legacy MPI code into a multithreaded environment where overlapping is achieved.

Keywords: User level threads; Parallel iterative reconstruction algorithms; Electron tomography.

1 Introduction

Cluster computing is a cost-effective solution for supercomputing, based on the usage of commodity hardware and standard software components. Parallel applications developed using message passing libraries such as MPI make intensive use of the concept of process as a building block, these applications therefore lack flexibility. Applications based on processes are monolithic computing entities, dulling important capabilities such as awareness of communication latencies because only one control stream is possible per process. Therefore when a process yields due a communication phase there will be no computation associated to that process. Latency hiding is up to the programmer.

Multithreaded programming provides a way to divide a program into different flows of control. Moreover using user level threads[1], [2] in places where concurrence can be exploited would allow us to achieve latency hiding, better scalability, skills to avoid overhead on processors due to faster context switching and abilities to migrate threads for load balancing purposes.

As opposed to the process monolithic approach, languages based on multithreaded and concurrent constructs allow easily more than one active flow of control within a process. Each active thread has even its own stack area and is executed concurrently under the control of a thread scheduler. Adaptive MPI

* Work supported through grants MEC-TIN 2005-00447 and JA-P06-TIC-01426.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 81–88, 2007.

(AMPI) [3] embeds MPI processes into user level threads, a more efficient technique in aspects like context switching, migration of tasks, etc.

Our application addresses the tomographic reconstruction problem using iterative methods. These methods are far superior to the standard technique, weighted backprojection, in terms of reconstruction quality [4]. However, they are much more computationally demanding. The capability of representing the density function in the volume by means of basis functions more general than the traditional voxels, is doubtlessly one of their main advantages. Research carried out during the 1990s [5], concluded that spherically symmetric volume elements (blobs) yield better reconstructions than voxels, as assessed in the fields of medicine [6] and electron tomography [4]. As drawbacks blobs lack resources when compared to voxel's computing needs.

In this work, we present and evaluate a parallel implementation of blob-based iterative reconstruction methods using a multithreaded approach with AMPI. The aim is to face the computational demands of these methods by exploiting concurrency and minimizing latency due to message passing. This paper is organized as follows. Next section analyzes the iterative reconstruction techniques. Section 3 describes the user level threads framework and Section 4 exposes how the tomographic reconstruction problem was parallelized based on the AMPI framework. Section 5 analyzes how this threaded implementation behaves when compared with its MPI version, by varying the thread's population. Finally the conclusions are exposed in last section.

2 Iterative Image Reconstruction

Let f be the function that represents the object to be reconstructed. Series expansion reconstruction methods assume that the 3D object or function, f, can be approximated by a linear combination of a finite set of known and fixed basis functions, b_j, with density x_j:

$$f(r, \theta_1, \theta_2) = \sum_{j=1}^{J} x_j \cdot b_j(r, \theta_1, \theta_2) \tag{1}$$

where (r, θ_1, θ_2) are spherical coordinates. The aim is to estimate the unknowns, x_j. These methods are based on an image formation model where the measurements depend linearly on the object in such a way that

$$y_i = \sum_{j=1}^{J} l_{i,j} \cdot x_j \tag{2}$$

where y_i denotes the i^{th} measurement of f and $l_{i,j}$ the value of the i^{th} projection of the j^{th} basis function. Under those assumptions, the image reconstruction problem can be modeled as the inverse problem of estimating the x_j's from the y_i's by solving the system of linear equations given by Eq. (2). Algebraic reconstruction techniques (ART) constitute one of the best known families of iterative algorithms to solve such systems [6]. The main drawback of iterative methods are their high computational requirements. These demands can be faced by means of

parallel computing and efficient reconstruction methods with fast convergence. Component averaging methods (CAV) have been developed recently [7] as efficient iterative algorithms for solving large and sparse systems of linear equations. In essence, these methods have been derived from ART methods, with the important innovation of a weighting related to the sparsity of the system. This component-related weighting provides the methods with a convergence rate that may be far superior to the ART methods. Assuming that the whole set of equations in the linear system -Eq. (2)- may be subdivided into B blocks each of size S, a generalized version of component averaging methods ($BICAV$) can be described via its iterative step from the k^{th} estimate to the $(k+1)^{th}$ estimate:

$$x_j^{k+1} = x_j^k + \lambda_k \cdot \sum_{s=1}^{S} \cdot \frac{y_i - \sum_{v=1}^{J} l_{i,v} x_v^k}{\sum_{v=1}^{J} s_v^b (l_{i,v})^2} \cdot l_{i,j} \tag{3}$$

where λ_k denotes the relaxation parameter; $b = (k \ mod \ B)$ and denotes the index of the block; $i = bS + s$ and represents the i^{th} equation of the whole system; and S_v^b denotes the number of times that the component x_v of the volume contributes with nonzero value to the equations in the b^{th} block. The processing of all the equations in one of the blocks produces a new estimate. All blocks are processed in one iteration of the algorithm. This technique produces iterates which converge to a weighted least squares solution of the system of equations provided that the relaxation parameters are within a certain range and the system is consistent [4]. Using higher number of blocks allows faster convergence.

Conceptually, block-iterative reconstruction algorithms may be decomposed into three subsequent stages: (i) computation of the forward-projection of the model; (ii) computation of the error between the experimental and the calculated projections; and (iii) refinement of the model by means of backprojection of the error. Aforementioned stages can be easily identified in Eq. (3). Those reconstruction algorithms pass through the stages for every block of equations and for every iteration.

2.1 Data Dependences Basis Functions

In the field of image reconstruction, the choice of the set of basis functions to represent the object to be reconstructed greatly influences the result of the algorithm. Overlapping spherically symmetric volume elements (blobs) with smooth transition to zero have been thoroughly investigated [5] as alternatives to voxels for image representation, concluding that the properties of blobs make them well suited to represent natural structures of all physical sizes. In addition, the use of blobs provides the reconstruction algorithm with an implicit regularization mechanism, very appropriate to work under noisy conditions.

A volume can be considered made up of 2D slices. When voxel basis functions are used, the slices are independent. However, the use of the blobs makes slices interdependent because of blob's overlapping nature. The interdependence depends on the blob extension.

3 User-Level Threads Framework

We have implemented the parallel iterative reconstruction method adapting legacy MPI code into a multithreaded environment by means of Adaptive MPI. AMPI is a virtualized implementation of MPI in which several MPI processes can efficiently share a single processor. This virtualization is achieved by making each traditional MPI process a migratable "user-level" thread, which can be thought of as a virtual processor [8], [9] (vp from now).

AMPI embodies a MPI process within an user level thread. Each thread is now a virtual processor, many AMPI virtual processors can coexist in the same processor. AMPI threads are lightweight and are managed by the Converse runtime [10] and not by OS routines. Typically user level threads are implemented entirely in user's space. As a result, context switching between user level threads in a process does not require any interaction with the kernel at all and is therefore extremely efficient: a context switch can be performed by locally saving the CPU registers used by the currently executing user level thread and loading the registers required by the user level thread to be executed. Since scheduling occurs in user's space, the scheduling policy provided by Converse can be easily tailored to the requirements of the program's workload. In MPI programming, having to block the CPU and wait for communication to complete can result in inefficiency. However, communication between AMPI threads is carried out through asynchronous calls, which do not block the objects involved in the communication. To get this, Converse maintains a couple of special data structures in each processor. One of them is the *sent messages* table which maintains a list of sent messages that are still pending for delivery, the other data structure is the *threads table* containing threads waiting for a message. When the processor gets an incoming message, it checks which thread is waiting for it, if any, and activates it. When a thread expresses the intention to receive a message, if the message is not ready, the thread will then be inserted into the threads table and put to sleep, and it will be tagged so that it can be identified and activated as soon as the message arrives.

4 Parallel Iterative Reconstruction Method

The block iterative version of the component averaging methods, BICAV, has been parallelized following the Single Program Multiple Data (SPMD) approach [4]. The volume is decomposed into slabs of slices that will be distributed across the nodes (for MPI) or virtual processors (for AMPI), see Fig.1 right, where slices are sketched with columns. The virtual processors can then process their own data subdomain, however, the interdependence among neighbor slices due to the blob extension makes necessary the inclusion of redundant slices into the slabs (see columns in light gray in Fig.1 right). In addition, there must be a proper exchange of information between neighbor nodes to compute the forward-projection or the error back-projection for a given slab and to keep redundant slices updated [4] (see Fig.1 left). These communication points constitute synchronization points in every pass of the algorithm in which the nodes must wait

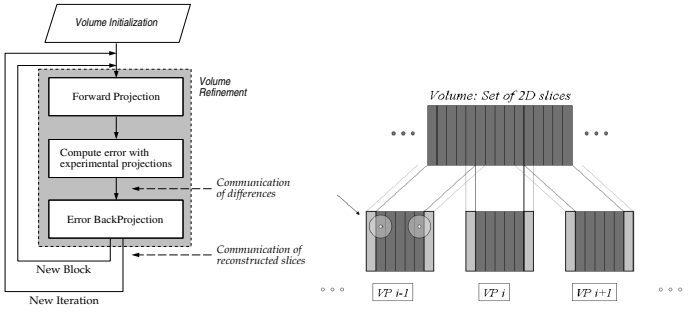


Fig. 1. Flow of the iterative reconstruction algorithm (left). Data decomposition (right).

for the neighbors. The amount of communications is proportional to the number of blocks and iterations.

In any parallelization project where communication between nodes is involved, latency hiding becomes an issue. That term stands for overlapping communication and computation so as to keep the processor busy while waiting for the communications to be completed.

Our improvement proposal is based upon AMPI user level thread's characteristic, used to achieve such latency hiding [3] by providing computation-communication overlapping. AMPI user level threads, -which embodies a classical MPI process- can switch from one thread to another when the running thread enters into one of the synchronization points. So while one thread is waiting, another one can take idle cpu cycles to progress with the application. Performance improvement depends on the algorithm used, the underlying network architecture and the system support for the context switching. As aforementioned, reconstruction yields better results when the number of blocks are increased. However, this rises communication needs, thereby prompting a performance loss. Now, with AMPI, each slab is embodied into a thread, so each node will be assigned as many slabs as threads. Communication patterns among neighboring nodes will maintain the same as each node will still communicate the same number of boundary slices in either AMPI or MPI implementations. As a remark, AMPI allows sleeping threads to come into action whenever some communication operation is in process, therefore performance is improved.

5 Results

Two different computing platforms were used to evaluate our reconstruction algorithm. One of the platforms used was NCSA's Linux cluster, TUNGSTEN, which owns a Dell poweredge 1750 server as front-end with 3GB ECC DDR SDRAM and compute nodes based on the Intel Xeon 3.2 GHz (32 bits), 512 KB L2 Cache and 1MB L3 Cache, all nodes interconnected with 1 Gb Ethernet Myrinet 2000. The second platform was our research group cluster, VERMEER, which owns

as a front-end a dual Pentium IV XEON 3.2 Ghz, 512KB Cache node with Gentoo Linux, Kernel 2.4.32 SMP and two Pentium IV XEON 3.06 Ghz, 512KB L2 Cache per computing node with 2GB of ECC DDR SDRAM, connected via 1Gb Ethernet. We were interested in the gain obtained by the multithreaded implementation with automatic overlapping of computation/communication, compared to the MPI version with the programmed latency hiding technique included. Scaling tests were carried out on both systems varying the number of threads per processor and the number of processors, for both versions AMPI and MPI. Two types of experiments were carried out to assess the performance of our multithreaded approach compared to the MPI implementation. In the first experiment, we aimed to measure the net gain by the implicit latency hiding mechanism provided in the AMPI implementation. This experiment was performed on the Tungsten cluster, using a dataset for a reconstruction of a volume of 256x256x256 from 70 projection images of 256x256 with the BICAV algorithm with a number of blocks of K=70. The number of processors was varied from 1 to 64, and the speedup was computed using (a) one thread per processor and (b) two threads per processor.

Fig. 2 clearly shows that the use of two threads per processor allows fully exploitation of the thread concurrency and that the implicit latency hiding succeeds in increasing the speedup, especially as the number of processors increases.

The second experiment was intended to quantify the influence of the number of blocks over the global performance. As the number of blocks increases, the convergence of the algorithm is faster, but the amount of communications also increases. This scenario harms the MPI version whereas AMPI is expected to keep good performance. This test was performed on the Vermeer cluster using up to 32 physical processors. Two datasets with different problem sizes were used. First, a dataset for a reconstruction of a volume of 256x256x256 from 256 projection images of 256x256. Second, a dataset for a reconstruction of a volume of 512x512x512 from 512 projection images of 512x512. The BICAV algorithms,

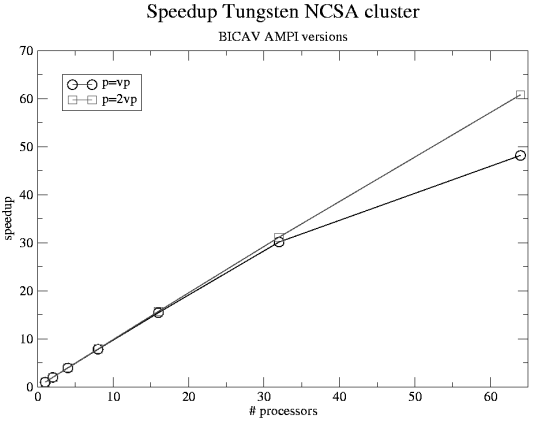


Fig. 2. Speedups obtained for BICAV in Tungsten cluster

in MPI and multithread AMPI versions, were executed with the two datasets. Two different numbers of blocks of equations K were used, one of them with the highest number of blocks for convergence purposes in BICAV and the other with an intermediate value.

Fig. 3 shows the gain obtained by the multithreaded of BICAV with 128 virtual processors, for two numbers of blocks K and for the two datasets. The speedup curves clearly show that the application implemented with MPI loses the linear speedup when using more than eight processors. However, AMPI keeps speedup almost linear up to 32 processors. Therefore, thread switching succeeds in maintaining the speedup optimal.

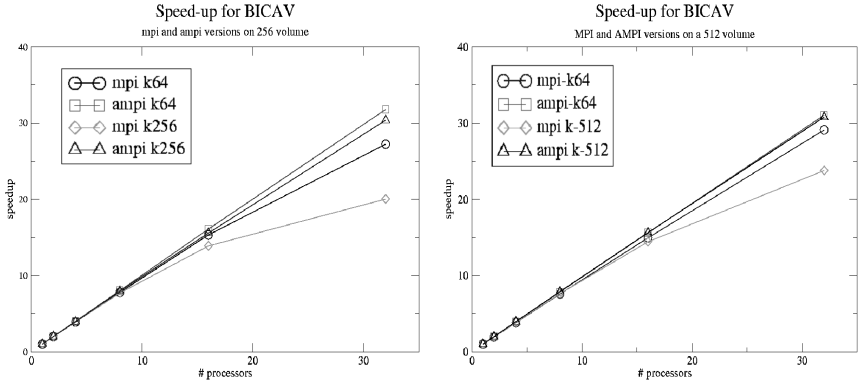


Fig. 3. Speedup in Vermeer cluster for 256 volume (left) and 512 volume (right)

Table 1. CPU and Wall time differences. Concurrency (k256-512)

	K 256 (volume 256)						K 512 (volume 512)					
	MPI			AMPI			MPI			AMPI		
Procs	CPU	WALL	%	CPU	WALL	%	CPU	WALL	%	CPU	WALL	%
2	991	1021	2.9	1012	1013	0.1	16749	17217	2.8	16989	16988	0.0
4	502	520	3.5	509	509	0	8432	8705	3.2	8479	8481	0.0
8	251	265	5.6	257	257	0	4222	4418	4.7	4251	4263	0.3
16	126	147	17.1	130	131	0.8	2128	2334	9.7	2138	2154	0.7
32	63	192	62.4	67	68	1.5	1071	1417	32.3	1092	1094	0.2

Table 1 presents the relative difference (columns %) among cputime and walltime for both problem sizes. This table shows the cases of the highest communication for both volumes (K=256 and K=512, respectively). It is easy to see how for AMPI walltime and cputime are almost the same, which means that the cpu was mostly in use and not idle. In contrast, the differences from the MPI version may turn out to be significant, especially for increasing number of processors. Taking into account that there are neither I/O operations nor function/method calls apart from those related to the own algorithm, we can conclude that for our multithreaded version for BICAV algorithm the concurrence is therefore seized at maximum.

6 Discussion and Conclusions

We presented a study about how multithreading can exploit latency hiding for grand-challenge applications like BICAV tomographic reconstruction. We based our study on AMPI supported by a powerful runtime called Converse that allows automatic user level thread switching avoiding idle cpu cycles. Through experiments we illustrated how one of our scientific iterative applications (*BICAV*) was directly benefited as the difference among cpu and walltime per iteration was nearly zero in contrast to the version based on MPI which did not respond so well. We further analyzed the performance of this technique while varying parameters as the number of physical processors and the population of virtual processors (threads) and we can state that the threaded version of *BICAV* scaled significantly better than the MPI version. From these experiments we conclude that user level threads are an attractive approach for programming parallel scientific applications, which can be benefited from having multiple flows of control.

References

1. Oikawa, S., Tokuda, H.: Efficient Timing Management for User-Level Real-Time Threads. In: Proceedings of the IEEE Real-Time Technology and Applications Symposium, pp. 27–32. IEEE, Los Alamitos (1995)
2. Price, G.W., Lowenthal, D.K.: A comparative analysis of fine-grain threads packages. Journal of Parallel and Distributed Computing 63, 1050–1063 (2003)
3. Huang, C., Lawlor, O., Kale, L.V.: Adaptive MPI. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004)
4. Fernández, J., Lawrence, A.F., Roca, J., García, I., Ellisman, M.H., Carazo, J.M.: High performance electron tomography of complex biological specimens. Journal of Structural Biology 138, 6–20 (2002)
5. Matej, S., Lewitt, R., Herman, G.: Practical considerations for 3-D image reconstruction using spherically symmetric volume elements. IEEE Trans. Med. Imag. 15, 68–78 (1996)
6. Herman, G.: Algebraic reconstruction techniques in medical imaging. In: Leondes, C. (ed.) Medical Imaging. Systems: Techniques and Applications, pp. 1–42. Gordon and Breach, New York (1998)
7. Censor, Y., Gordon, D., Gordon, R.: BICAV: a block-iterative, parallel algorithm for sparse systems with pixel-related weighting. IEEE Trans. Med. Imag. 20, 1050–1060 (2001)
8. Kale, L.V.: The Virtualization Model of Parallel Programming: Runtime Optimizations and the State of Art. In: Los Alamos Computer Science Institute Symposium (LACSI 2002). Alburquerque (2002)
9. Kale, L.V., Lawlor, O.S., Bhandarkar, M.: Parallel Objects: Virtualization and in-Process Components. In: Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02), 16th Annual ACM International Conference on Supercomputing (ICS'02), ACM Press, New York (2002)
10. Bhandarkar, M.A.: Charisma: A Component Architecture for Parallel Programming. PhD thesis, University Illinois at Urbana-Champaign (2002)

Parallelizing Dense Linear Algebra Operations with Task Queues in `llc`*

Antonio J. Dorta[1], José M. Badía[2], Enrique S. Quintana-Ortí[2], and Francisco de Sande[1]

[1] Depto. de Estadística, Investigación Operativa y Computación
Universidad de La Laguna, 38271–La Laguna, Spain
{ajdorta,fsande}@ull.es
[2] Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I, 12.071–Castellón, Spain
{badia,quintana}@icc.uji.es

Abstract. `llc` is a language based on C where parallelism is expressed using compiler directives. The `llc` compiler produces MPI code which can be ported to both shared and distributed memory systems.

In this work we focus our attention in the `llc` implementation of the *Workqueuing Model*. This model is an extension of the OpenMP standard that allows an elegant implementation of irregular parallelism. We evaluate our approach by comparing the OpenMP and `llc` parallelizations of the symmetric rank-k update operation on shared and distributed memory parallel platforms.

Keywords: MPI, OpenMP, Workqueuing, cluster computing, distributed memory.

1 Introduction

The advances in high performance computing (HPC) hardware have not been followed by the software. The tools used to express parallel computations are nowadays one of the major obstacles for the massive use of HPC technology. Two of these tools are MPI [1] and OpenMP [2]. Key advantages of MPI are its portability and efficiency, with the latter strongly influenced by the control given to the programmer of the parallel application. However, a deep knowledge of low-level aspects of parallelism (communications, synchronizations, etc.) is needed in order to develop an efficient MPI parallel application.

On the other hand, OpenMP allows a much easier implementation. One can start from a sequential code and parallelize it incrementally by adding compiler directives to specific regions of the code. An additional advantage is that it

* This work has been partially supported by the EC (FEDER) and the Spanish MEC (Plan Nacional de I+D+I, TIN2005-09037-C02).

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 89–96, 2007.

follows the sequential semantic of the program. The main drawback of OpenMP is that it only targets shared memory architectures.

As an alternative to MPI and OpenMP, we have designed `llc` [3] to exploit the best features of both approaches. `llc` shares the simplicity of OpenMP: we can start from a sequential code and parallelize it incrementally using OpenMP and/or `llc` directives and clauses. The code annotated with parallel directives is compiled by `llCoMP`, the `llc` compiler-translator, which produces an efficient and portable MPI parallel source code, valid for both shared and distributed memory architectures. An additional advantage of `llc` is that all the OpenMP directives and clauses are recognized by `llCoMP`. Therefore, we have three versions in the same code: sequential, OpenMP and `llc`/MPI, and we only need to choose the proper compiler to obtain the appropriate binary.

Different directives have been designed in `llc` to support common parallel constructs in the past as *forall*, *sections*, and *pipelines* [4,5]. In previous studies [4] we have investigated the implementation of *Task Queues* in `llc`. In this paper we focus our attention in the last feature added to `llc`: the support for the *Workqueuing Model* using *Task Queues* [6]. In order to do so, we explore the possibilities of parallelizing (dense) linear algebra operations, as developed in the frame of the *FLAME* (Formal Linear Algebra Method Environment) project [7].

The rest of the paper is organized as follows. In Section 2 we present the *symmetric rank-k update* (`SYRK`) operation as well as a FLAME code for its computation. Section 3 reviews the parallelization of this code using OpenMP and `llc`. Experimental results for both OpenMP and `llc` codes are reported and discussed in Section 4. Finally, Section 5 offers some concluding remarks and hints on future research.

2 The `SYRK` Operation

The `SYRK` operation is one of the *Basic Linear Algebra Subprograms* (BLAS) [8] most often used. It plays an important role, e.g., in the formation of the normal equations in linear least-squares problems and the solution of symmetric positive definite linear systems via the Cholesky factorization [9]. The operation computes the lower (or upper) triangular part of the result of the matrix product $C := \beta C + \alpha A A^T$, where C is an $m \times m$ symmetric matrix, A is an $m \times k$ matrix, and α, β are scalars.

Listing 1 presents the FLAME code for the `SYRK` operation [10]. The partitioning routines (`FLA_Part_x`, `FLA_Repart_x_to_y` and `FLA_Cont_with_x_to_y`) are indexing operations that identify regions (blocks) into the matrices but do not modify their contents. Thus, e.g., the invocation to `FLA_Part_2x1` in lines 7–8 "divides" matrix (object) `A` into two submatrices (blocks/objects), `AT` and `AB`, with the first one having 0 rows. Then, at each iteration of the loop, certain operations are performed with the elements in these submatrices (routines `FLA_Gemm` and `FLA_Syrk`). More details can be consulted in [7].

```
int FLA_Syrk_ln_blk_var1_seq (FLA_Obj alpha, FLA_Obj A,
                              FLA_Obj beta,  FLA_Obj C, int nb_alg) {
  FLA_Obj AT, AB,       CTL, CBL, CTR, CBR,
          A0, A1, A2,   C00, C01, C02, C10, C11, C12, C20, C21, C22;
  int     b;

  FLA_Part_2x1(A, &AT,
                  &AB, 0, FLA_TOP);
  FLA_Part_2x2(C, &CTL, &CTR,
                  &CBL, &CBR, 0, 0, FLA_TL);

  while (FLA_Obj_length(AT) < FLA_Obj_length(A)){
    b = min(FLA_Obj_length(AB), nb_alg);
    FLA_Repart_2x1_to_3x1(AT, &A0,
                              &A1,
                           AB, &A2, b, FLA_BOTTOM);
    FLA_Repart_2x2_to_3x3(CTL, CTR, &C00, &C01, &C02,
                                    &C10, &C11, &C12,
                           CBL, CBR, &C20, &C21, &C22, b, b, FLA_BR);
    /*------------------------------------------------------------*/
    /* C10 := C10 + A1 * A0' */
    FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE, alpha, A1, A0,
                                              beta,  C10, nb_alg);
    /* C11 := C11 + A1 * A1' */
    FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, alpha, A1,
                                                     beta,  C11,nb_alg);
    /*------------------------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1(&AT, A0,
                                  A1,
                             &AB, A2, FLA_TOP);
    FLA_Cont_with_3x3_to_2x2(&CTL, &CTR, C00, C01, C02,
                                         C10, C11, C12,
                             &CBL, &CBR, C20, C21, C22, FLA_TL);
  }
  return FLA_SUCCESS;
}
```

Listing 1. FLAME code for the SYRK operation

3 Parallelization of the SYRK Operation

A remarkable feature of FLAME is its capability for hiding intricate indexing in linear algebra computations. However, this feature is a drawback for the traditional OpenMP method to obtain parallelism from a sequential code, based on exploiting the parallelism of `for` loops. Thus, the OpenMP approach requires loop indexes for expressing parallelism which are not available in FLAME codes.

Task Queues [6] have been proposed for adoption in OpenMP 3.0 and are currently supported by the Intel OpenMP compilers. Their use allows an elegant implementation of loops when the space iteration is not known in advance or, as in the case of FLAME code, when explicit indexing is to be avoided.

3.1 OpenMP Parallelization

The parallelization of the SYRK operation using the Intel implementation of *Task Queues* is described in [10]. The Intel extension provides two directives to specify tasks queues. The `omp parallel taskq` directive specifies a parallel region where tasks can appear. Each task found in this region will be queued for later

computation. The `omp task` identifies the tasks. For the `SYRK` operation, the first clause is used to mark the `while` loop (line 12 in Listing 1), while the second one identifies the invocations to `FLA_Gemm` and `FLA_Syrk` as tasks (lines 21–26 in Listing 1). Listing 2 shows the parallelization using `taskq` of the loop in the

```
1  #pragma intel omp parallel taskq{
2    while (FLA_Obj_length(AT) < FLA_Obj_length(A)){
3    ...
4      #pragma intel omp task captureprivate(A0, A1, C10, C11){
5        /* C10 := C10 + A1 * A0' */
6        FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE, alpha, A1, A0,
7                                                  beta,  C10, nb_alg);
8        /* C11 := C11 + A1 * A1' */
9        FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, alpha, A1,
10                                                      beta,  C11,nb_alg);
11     }
12   ...
13   }
14 }
```

Listing 2. FLAME code for the `SYRK` operation parallelized using OpenMP

FLAME code for the `SYRK` operation. The directive `omp task` that appears in line 4 is used to identify the tasks. Function calls to `FLA_Gemm` and `FLA_Syrk` are in the scope of the `taskq` directive in line 1 and, therefore, a new task that computes both functions is created at each iteration of the loop. The first of these functions computes $C_{10} := C_{10} + A_1 A_0^T$, while the second one computes $C_{11} := C_{11} + A_1 A_1^T$. All the variables involved in these computations have to be private to each thread (A_0, A_1, C_{10}, and C_{11}), ant thus they must be copied to each thread during execution time. The `captureprivate` clause that complements the `omp parallel task` directive serves this purpose.

3.2 `llc` Parallelization

In this section we illustrate the use of `llc` to parallelize the `SYRK` code. Further information about the effective translation of the directives in the code to MPI can be found in [4]. The parallelization using `llc` resembles that carried out using OpenMP, with a few differences that are illustrated in the following. After identifying the task code, we annotate the regions using `llc` and/or OpenMP directives. All the OpenMP directives and clauses are accepted by `llCoMP`, though not all of them have meaning and/or effect in `llc` [4].

We will start from the OpenMP parallel code shown in Listing 2 and we will add the necessary `llc` directives in order to complete the `llc` parallelization. The OpenMP `captureprivate` clause has no sense in `llc`, because `llCoMP` produces a MPI code where each processor has its private memory. (`llc` follows the OTOSP model [3], where all the processors on the same group have the same data in their private memories.) Unlike OpenMP, in `llc` all the variables are private by default, and we have to use `llc` directives to specify shared data. Listing 3 shows the parallelization of the FLAME code for the `SYRK` operation using `llc`.

```
1  #pragma intel omp task
2  #pragma llc task_master_data(&A0.m, 1, &A1.offm, 1, &A1.m, 1)
3  #pragma llc task_master_data(&C11.offm,1,&C11.offn,1,&C11.m,1,&C11.n,1)
4  #pragma llc task_master_data(&C10.offm,1,&C10.offn,1,&C10.m,1,&C10.n,1)
5  #pragma llc task_slave_set_data(&A1.base,1,A.base,&A0.base,1,A.base)
6  #pragma llc task_slave_set_data(&C11.base,1,C.base,&C10.base,1,C.base)
7  #pragma llc task_slave_set_data(&A0.offm,1,A.offm,&A0.offn,1,A.offn,&A0.
       n,1,A.n)
8  #pragma llc task_slave_set_data(&A1.offn,1,A.offn,&A1.n,1,A.n)
9  #pragma llc task_slave_rnc_data ((C10.base->buffer+((C10.offn*C10.base->
       ldim+C10.offm)*sizeof(double))), (C10.m * sizeof(double)), ((C10.
       base->ldim - C10.m) * sizeof(double)), C10.n)
10 #pragma llc task_slave_rnc_data ((C11.base->buffer+((C11.offn*C11.base->
       ldim+C11.offm)*sizeof(double))), (C11.m * sizeof(double)), ((C11.
       base->ldim - C11.m) * sizeof(double)), C11.n)
11   {
12   /* C10 := C10 + A1 * A0' */
13   FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE, alpha, A1, A0,
14                                              beta, C10, nb_alg);
15   /* C11 := C11 + A1 * A1' */
16   FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, alpha, A1,
17                                                  beta, C11, nb_alg);
18   }
```

Listing 3. FLAME code for the SYRK operation parallelized using llc

A first comparison of Listings 2 and 3 shows an apparent increase in the number of directives when llc is used. However, note that only three directives are actually needed, but we split those in order to improve the readability. Although llc code can be sometimes as simple as OpenMP code (see, e.g., [4]), here we preferred to use an elaborated algorithm to illustrate how llc overcomes difficulties that usually appear when targeting parallel distributed memory architectures: references to specific data inside a larger data structure (submatrices instead of the whole matrix), access to non-contiguous memory locations, etc.

In the llc implementation of *Task Queues*, a master processor handles the task queue, sends subproblems to the slaves, and gathers the partial results to construct the solution. Before the execution of each task, the master processor needs to communicate some initial data to the slaves, using the llc task_master_data directive. As the master and slaves processors are on the same group, they have the same values in each private memory region. Exploiting this, the master processor only sends those data that have been modified. With this approach the number of directives to be used is larger than in the OpenMP case, but the amount of communications is considerably reduced.

The master needs to communicate to each slave the offset and number of elements of the objects A_0, A_1, C_{10}, and C_{11} (lines 2–4). After each execution, the slave processors "remember" the last data used. To avoid this, we employ the llc task_slave_set_data directives in lines 5–8 that initialize the variables before each task execution to certain fixed values (with no communications involved).

The code inside the parallel task computes $C_{10} := C_{10} + A_1 A_0^T$ and $C_{11} := C_{11} + A_1 A_1^T$. The slaves communicate to the master the results obtained (C_{10} and C_{11}). These data are not stored in contiguous memory positions and therefore can not be communicated as a single block. However, the data follow a regular

pattern and can be communicated using the `llc task_slave_rnc_data` directive (lines 9–10). This directive specifies *r*egular *n*on-*c*ontiguous memory locations.

4 Experimental Results

All the experiments reported in this section for the `SYRK` operation ($C := C + AA^T$, with an $m \times m$ matrix C and an $m \times k$ matrix A) were performed using double-precision floating point arithmetic. The results correspond to the codes that have been illustrated previously in this paper (FLAME Variant 1 of the `SYRK` operation, Var1) as well as a second variant (Var2) for the same operation [10].

Three different platforms were employed in the evaluation, with the common building block in all these being an Intel Itanium2 1.5GHz processor. The first platform is a shared-memory (SM) Bull NovaScale 6320 with 32 processors. The second platform is a SM SGI Altix 250 with 16 processors. The third system is a hybrid cluster composed of 9 nodes connected via a 10 Gbit/s InfiniBand switch; each node is a SM architecture with 4 processors, yielding a total of 36 processors in the system. An extensive experimentation was performed to determine the best block size (parameter `nb_alg` in the algorithms) for each variant and architecture. Only those results corresponding to the optimal block size (usually, around 96) are reported next.

The OpenMP implementations were compiled with the Intel C compiler, while the `llc` binaries were produced with `llCoMP` combined with the `mpich` implementation of MPI on the SGI Altix and hybrid cluster, and MPIBull-Quadrics 1.5 on the NovaScale server.

The goal of the experiments on SM platforms is to compare the performance of the `SYRK` implementation in OpenMP and `llc`. The results on the hybrid system are presented to demonstrate that high performance can be also achieved when the portability of `llc` is exploited.

Table 1 reports the results for the `SYRK` codes. In particular, the second row of the table shows the execution time of the sequential code, while the remaining rows illustrate the speed-up of the OpenMP and `llc` parallelizations on the SGI Altix and the Bull NovaScale.

The results show a similar performance for OpenMP and our approach on both architectures. OpenMP obtains a higher performance than `llc` when the number of processors is small. The reason for this behavior is that in the `llc` implementation one of the processors acts as the master. As the number of processors grows, the speed-up of `llc` increases faster than that of OpenMP. When the number of processors is large, `llc` yields better performance than OpenMP because it is less affected by memory bandwidth problems. The second variant of the algorithm exhibits a better performance than the first one, because it generates a larger number of tasks with finer granularity during the computations following a bidimensional partitioning of the work; see [10].

Figure 1 shows the speed-up obtained on the hybrid system. Again the second variant exhibits a better performance, and a maximum speed-up slightly above 25 is attained using 36 processors.

Table 1. Sequential time and speed-up obtained on the SM platforms for Variants 1 and 2 of the SYRK operation for both OpenMP and llc. For the Bull NovaScale 6320 (Bull), m=10000 and k=7000. For the SGI Altix 250 (SGI), m=6000 and k=3000.

#Proc.	Var1 SGI		Var1 Bull		Var2 SGI		Var2 Bull	
seq.	*19.0 sec.*		*176.5 sec.*		*19.0 sec.*		*176.5 sec.*	
–	omp	llc	omp	llc	omp	llc	omp	llc
3	2.13	1.58	2.75	1.85	2.83	1.89	2.94	1.98
4	2.85	2.22	3.48	2.65	3.72	2.82	3.84	2.96
6	3.97	3.49	4.25	4.16	5.51	4.72	5.34	4.91
8	4.60	4.68	5.16	5.59	7.16	6.52	6.74	7.21
10	5.78	5.70	6.83	6.98	8.82	8.33	8.16	8.62
12	6.76	7.41	7.34	7.81	10.24	10.09	9.53	10.65
14	6.69	7.81	7.93	8.90	11.67	11.79	9.37	13.20
16	7.41	9.02	8.61	9.35	12.71	13.62	9.56	13.76

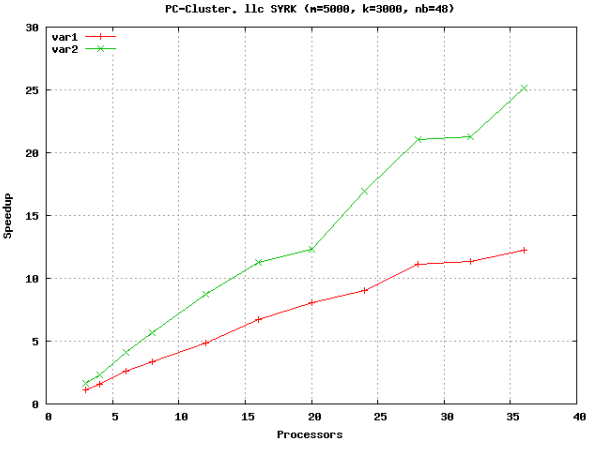


Fig. 1. Speed-up on the hybrid system for Variants 1 and 2 of the SYRK operation parallelized using llc. On this system, m=5000 and k=3000.

5 Conclusions and Future Work

llc is an language based on C that, given a sequential code annotated with directives and using the llCoMP translator-compiler, produces MPI parallel code. llc combines the high productivity in code development of OpenMP with the high performance and the portability of MPI.

In this paper we have evaluated the performance of the *Task Queues* implementation in llc using FLAME codes for the SYRK operation. We have shown that the llc directives facilitate optimization and tuning. The additional complexity

introduced in the `llc` version with respect to the OpenMP version is clearly paid off by the portability of the code. The performance achieved with our approach is comparable to that obtained using OpenMP. Taking into account the smaller effort to develop codes using `llc` compared with a direct MPI implementation, we conclude that `llc` is appropriate to implement some classes of parallel applications.

Work in progress concerning this topic includes the following:

- To study other variants and parallelization options for the `SYRK` operation, such as using two tasks per iteration or splitting the `while` loop.
- To study other FLAME operations. We are currently working on the matrix-vector product.
- To apply our approach to other scientific and engineering applications.
- To extend the computational results to other machines and architectures.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. University of Tennessee, Knoxville, TN (1995), `http://www.mpi-forum.org/`
2. OpenMP Architecture Review Board: OpenMP Application Program Interface v. 2.5 (2005)
3. Dorta, A.J., González, J.A., Rodríguez, C., de Sande, F.: llc: A parallel skeletal language. Parallel Processing Letters 13(3), 437–448 (2003)
4. Dorta, A.J., Lopez, P., de Sande, F.: Basic skeletons in llc. Parallel Computing 32(7–8), 491–506 (2006)
5. Dorta, A.J., Badía, J.M., Quintana, E.S., de Sande, F.: Implementing OpenMP for clusters on top of MPI. In: Di Martino, B., Kranzlmüller, D., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3666, pp. 148–155. Springer, Heidelberg (2005)
6. Shah, S., Haab, G., Petersen, P., Throop, J.: Flexible control structures for parallelism in OpenMP. Concurrency: Practice and Experience 12(12), 1219–1239 (2000)
7. Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Ortí, E.S., van de Geijn, R.A.: The science of deriving dense linear algebra algorithms. ACM Trans. on Mathematical Software 31(1), 1–26 (2005)
8. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for fortran usage. ACM Trans. Math. Softw. 5(3), 308–323 (1979)
9. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore, MD (1996)
10. Van Zee, F., Bientinesi, P., Low, T.M., van de Geijn, R.A.: Scalable parallelization of FLAME code via the workqueuing model (ACM Trans. on Mathematical Software) (to appear) Electronically, available at `http://www.cs.utexas.edu/users/flame/pubs.html`

ParaLEX: A Parallel Extension for the CPLEX Mixed Integer Optimizer*

Yuji Shinano[1] and Tetsuya Fujie[2]

[1] Division of Systems and Information Technology, Institute of Symbiotic Science and Technology, Tokyo University of Agriculture and Technology, 2-24-16, Naka-cho, Koganei-shi, Tokyo 184-8588, Japan
yshinano@cc.tuat.ac.jp
[2] School of Business Administration, University of Hyogo, 8-2-1, Gakuen-nishimachi, Nishi-ku, Kobe 651-2197, Japan
fujie@biz.u-hyogo.ac.jp

Abstract. The ILOG CPLEX Mixed Integer Optimizer is a state-of-the-art solver for mixed integer programming. In this paper, we introduce ParaLEX which realizes a master-worker parallelization specialized for the solver on a PC cluster using MPI. To fully utilize the power of the solver, the implementation exploits almost all functionality available in it. Computational experiments are performed for MIPLIB instances on a PC cluster composed of fifteen 3.4GHz pentiumD 950 (with 2G bytes RAM) PCs (running a maximum of 30 CPLEX Mixed Integer Optimizers). The results show that ParaLEX is highly effective in accelerating the solver for hard problem instances.

Keywords: Mixed Integer Programming, Master-Worker, Parallel Branch-and-cut.

1 Introduction

The ILOG CPLEX Mixed Integer Optimizer [6] is one of the most successful commercial codes for MIP (Mixed Integer Programming). MIP problem is to optimize (minimize or maximize) a linear function subject to linear inequalities and/or linear equalities with the restriction that some or all of the variables must take integer values. MIP has a wide variety of industrial, business, science and educational applications. In fact, recent remarkable progress of MIP optimizers, including CPLEX, leads to the increased importance of MIP models. CPLEX has continued to incorporate computational improvements into a branch-and-cut implementation, which results an efficient and robust code. It involves both standard and advanced techniques such as preprocessing, many kinds of cutting planes, heuristics, and strong branching. Due to limited space, we omitted MIP-related from this paper (see [8,11] for MIP models and algorithms). In [2], recent software systems for MIP are introduced. Many researchers have developed

* This work was partially supported by MEXT in Japan through Grants-in-Aid(18510118).

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 97–106, 2007.

parallelization frameworks of branch-and-bound and branch-and-cut algorithms, including ALPS/BiCeOS, PICO, SYMPHONY, BCP, PUBB. See recent surveys [3,4,9].

In this paper, we propose a master-worker parallelization of CPLEX, named ParaLEX (Parallel extension for CPLEX MIP optimizer). The main feature of our implementation is that it fully utilizes the power of CPLEX: The entire search tree (or, branch-and-cut tree) produced by ParaLEX is composed of subtrees produced by CPLEX on master and workers. ParaLEX has a simpler structure than our previous work of a parallelization of CPLEX using the PUBB2 framework [10].

2 ParaLEX

In this section, we introduce ParaLEX briefly. We had three major goals for the design of ParaLEX.

- Most of all the functionality of CPLEX must be available.
- Future versions of CPLEX must be "parallelizable" without any code modification needed.
- Parallel implementation must be as simple as possible.

To achieve these design goals, ParaLEX is

- composed of two types of solvers, each of which runs CPLEX with almost full functionality,
- is implemented in C++ but the most primitive CPLEX Callable Library is used, and
- is essentially a simple Master-Worker parallelization.

The branch-and-cut algorithm is an enumerative algorithm which repeatedly partitions a given problem instance into several subproblems. The algorithm therefore forms a tree called search tree or branch-and-cut tree. Subproblems will be also referred to as nodes in the subsequent of this paper. Similarly, a given problem instance will be also referred to as a root node. Next, we introduce the notion of the *ParaLEX instance* which comprises a preprocessed problem instance and global cuts applied to the root node. Preprocessing and generation of global cuts (or, cutting planes), both of which are applied to the root node, are quite effective in CPLEX in giving a tighter reformulation of MIP even though they are time consuming. It is certain that, if we drop these functionalities in parallelization, the solution time of MIP becomes greater than that by a sequential CPLEX solver. On the other hand, if these functionalities are performed from scratch on each PE (Processing Element), the parallelization cannot achieve high performance in general. ParaLEX instances are introduced based on the observation. A subproblem representation used in ParaLEX is the difference between a node LP object in the branch callback routine of CPLEX and the ParaLEX instance.

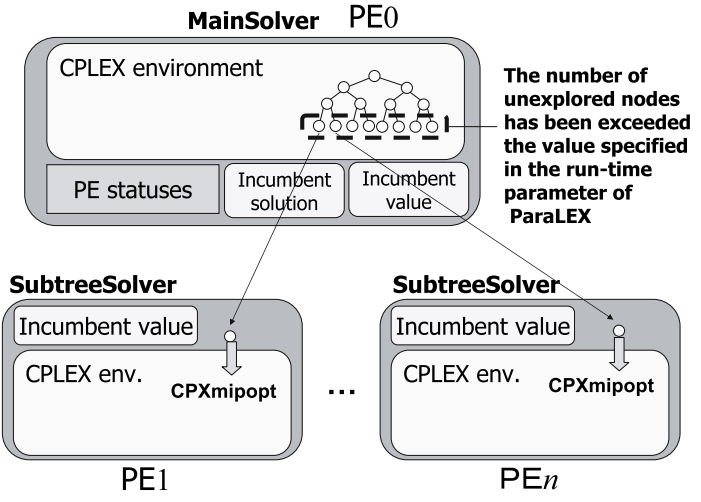


Fig. 1. Initialization phase of ParaLEX

ParaLEX is implemented as a SPMD-type MPI program[1] but is composed of the following two types of objects (solvers) depending on the rank of the process, which we call the *PE rank*.

MainSolver (PE rank $= 0$**).** This solver reads an original problem instance data, and then performs preprocessing, and generates global cuts to create the ParaLEX instance. It not only manages the PEs on the system but also solves nodes by applying the `CPXmipopt` function of CPLEX. It keeps an incumbent solution which is the best solution found so far among all the PEs and, at the end of computation, outputs an optimal solution.

SubtreeSolver (PE rank ≥ 1**).** This solver first receives the ParaLEX instance from the MainSolver. It also receives nodes from the MainSolver or other SubtreeSolvers. The nodes received are solved by CPLEX. If an improved incumbent solution is found in this solver, the solution is sent to the MainSolver.

At first, the MainSolver starts solving the ParaLEX instance by CPLEX. When the number of unexplored nodes in the CPLEX environment has been exceeded the threshold value given by the ParaLEX run-time parameter, the MainSolver starts distributing its unexplored nodes one by one to SubtreeSolvers. In our computational experiments reported in Section 3, we set the threshold parameter as $\min\{200, 5.0/(\text{average time to compute one node in the sequential run})\}$. Figure 1 shows the initialization phase.

When ParaLEX runs with several PEs, a search tree is partitioned into several subtrees produced by the corresponding PEs. Each subtree is maintained in the

[1] The Master-Worker program can also be implemented as an MPMD-type MPI program. However, we have been familiar with an SPMD implementation due to the requirement of MPI-1 functionality.

CPLEX environment of the corresponding PE (MainSolver or SubtreeSolver). In normal running situations, communications between the MainSolver and the SubtreeSolver and between the SubtreeSolvers are done in the branch callback routine of CPLEX. In this callback routine, the PE from which the callback is called sends a node to a PE if necessary. The node is then solved as a problem instance by CPLEX. Note that the node transferred is solved twice, once in the sender as a node and once in the receiver as a root node. In the receiver side, however, the root node is not solved from scratch, because its LP (Linear Programming) basis generated in the sender side is also transferred and is used as the starting basis. Though it is solved as a root node, it may be pruned without any branch by trying to apply extra cuts to the node. Eventually, it may lead to reduced total computation times.

The MainSolver maintains statuses of all the PEs. Each SubtreeSolver notifies to the MainSolver the number of unexplored nodes in the CPLEX environment and the best bound value among these nodes as the PE status, when the number of nodes processed from the previous notification has been exceeded the threshold value given by the ParaLEX run-time parameter. When a PE has finished solving an assigned node, it becomes an idle solver. If the MainSolver detects an idle solver, it sends a subproblem-transfer request message to a PE which has an unexplored node with the best bound value by referring to the PE statuses. There is a delay updating the PE statuses in the MainSolver. Therefore, the PE which receives the subproblem-transfer request may not have enough many nodes in its CPLEX environment. In such a case, the solver rejects the request. If the solver has more nodes than the threshold value given by the ParaLEX run-time parameter, it accepts the request and sends a node to the destination PE that is indicated in the request message. Note that at most one node can be transferred in one request message. Figure 2 shows a message sequence for the subproblem transfer.

The node to be transferred can be selected in the selection callback routine of CPLEX. When the node with the best bound transfer is specified in the ParaLEX run-time parameter, the best bound node is selected in this callback routine. The selection is done by a computationally intensive linear search of the unexplored nodes in the CPLEX environment. On the other hand, when a default selection is specified in this parameter, a node is selected according to the selection rule specified in the CPLEX run-time parameter. In this case, the unexplored nodes are arranged by the selection rule order in the CPLEX environment, and thus selection does not take a long time.

When an improved solution is found in the MainSolver, its incumbent value is sent to all the SubtreeSolvers. When an improved solution is found in the SubtreeSolver, the incumbent solution is sent to the MainSolver and the MainSolver sends the incumbent value to all the SubtreeSolvers. Improved incumbent solutions are detected in the incumbent callback routine of CPLEX and its notification is done in the branch callback of CPLEX.

The `CPXmipopt` function that solves a problem instance is suspended after the number of nodes specified in the ParaLEX run-time parameter has

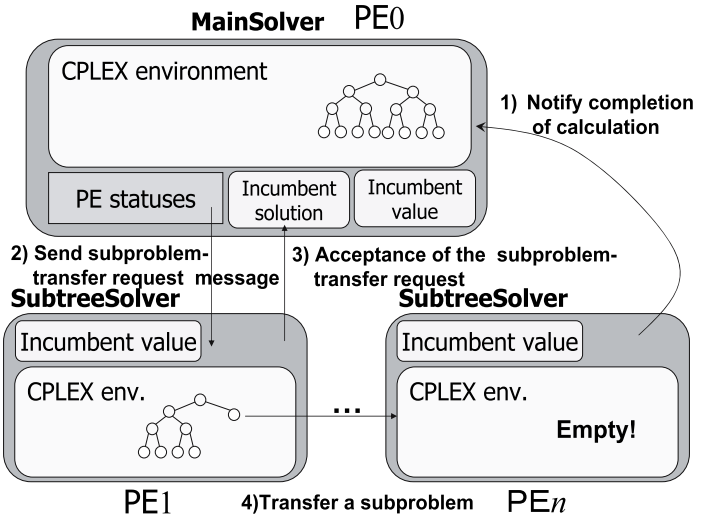


Fig. 2. Message sequence for the subproblem transfer

been processed. When suspended time comes, the latest incumbent value is set on CPLEX environment using the `CPXsetdblparam` function of CPLEX with `CPX_PARAM_CUTUP` (if the problem is minimization) or with `CPX_PARAM_CUTLO` (if the problem is maximization). After that, the suspension is resumed. Therefore, there is a delay in the notification of the incumbent value.

3 Computational Experiments

In this section, we report our computational results. We used a PC cluster composed of fifteen 3.4GHz pentiumD 950 (with 2Gbytes RAM) PCs connected with Gigabit Ethernet, where 30 CPLEX Mixed Integer Optimizers (version 10.1) were available. The MPI library used is mpich2-1.0.5p4. Problem instances were selected from the MIPLIB2003 library[2] [1] which is a standard test set to evaluate the performance of MIP optimizers.

We first discuss the effects of the ParaLEX run-time parameters described in Section 2.

- MIP emphasis indicator `CPX_PARAM_MIPEMPHASIS` (CPLEX) : This is a parameter prepared by CPLEX to tell the solver whether it should find a feasible solution with high quality or prove the optimality. Among several options, we selected the following ones, which are concerned with the optimality.
  - `CPX_MIPEMPHASIS_BALANCED` : Balance optimality and integer feasibility (default parameter of CPLEX).
  - `CPX_MIPEMPHASIS_OPTIMALITY` : Emphasizing optimality over feasibility.

[2] URL: http://miplib.zib.de

  - CPX_MIPEMPHASIS_BESTBOUND : Emphasizing moving best bound.
- Subproblem transfer mode (ParaLEX) : This is a parameter for PEs to transform a node.
  - best_bound_transfer : The best bound node is selected.
  - cplex_default_transfer : A node is selected according to the CPLEX run-time parameter.
- CPXmipopt interval (ParaLEX) : This is a parameter for PEs to specify the suspended time of CPXmipopt.
  - fixed_iter : Every time CPXmipopt terminates with the node limit of this value to check a subproblem-request from an another PE. In this paper, we set fixed_iter = 20.
  - fixed_time : By a sequential run, we estimate the number of nodes generated by CPXmipopt during the time fixed_time. Then, this estimated value is used for the node limit of CPXmipopt. In this paper, we set fixed_time = 1 (sec.).

Table 1 displays the parallel speedups obtained over 5 runs for the easy instances fast0507 and mas74 with possible combinations of the following two parameters, the Subproblem transfer mode and the CPXmipopt interval. CPX_PARAM_MIPEMPHASIS is set to CPX_MIPEMPHASIS_BALANCED. As the table shows, it is hard to determine the most suitable combination of the parameter values. On the other hand, cplex_default_transfer is competitive or better than best_bound_transfer, which indicates that a PE which receives a transferred node should continue the branch-and-cut search along with the parent PE (i.e., the PE which sends the node). Hence, we selected cplex_default_transfer for the subsequent computations. We observed that the effect of the CPXmipopt interval depends on the behavior of a sequential run of CPLEX since computing time per node varies considerably with problem instances. Hence, we decided to use fixed_time.

Table 2 shows the results for the CPX_PARAM_MIPEMPHASIS parameter. The results are obtained over 5 runs except that the noswot instance is examined over 1 run. We note that the computing time of sequential run varies with this parameter, and the most suitable parameter-choice also varies with the problem instance. From the table, we see that superlinear speedup results are obtained for several problem instances. In particular, noteworthy speedup is obtained for the noswot instance. We observed that superlinear speedup occurs when a good feasible solution is hardly obtained by CPLEX. In this case, a parallel search could find a good feasible solution faster than a sequential search. Since fast finding of a good feasible solution can lead to pruning of many unexplored nodes, the parallel search could generate a smaller search tree than the sequential search. Therefore, computation is performed to a smaller number of nodes by many PEs and, as a result, the superlinear speedup is obtained. We also observe that high speedup results are obtained for CPX_MIPEMPHASIS_OPTIMALITY.

Finally, we report the results for hard problem instances with 30 PEs. We continue to use the parameters cplex_default_transfer and fixed_time. CPX_PARAM_MIPEMPHASIS was determined by an observation of the behavior of upper and lower bounds within several hours from the beginning and by the information in the MIPLIB 2003 website.

Table 1. Parallel speedups for easy instances (`CPX_MIPEMPHASIS_BALANCED`)

	# of PEs					
	2	3	5	10	20	30
`fast0507`, `best_bound_transfer`, `fixed_iter` : 880.61 sec. (sequential)						
ave.	0.45	1.16	1.38	1.98	2.96	3.07
min.	0.45	1.13	1.37	1.97	2.93	3.04
max.	0.45	1.21	1.39	1.99	3.00	3.13
`fast0507`, `best_bound_transfer`, `fixed_time` : 880.61 sec. (sequential)						
ave.	0.99	1.04	1.33	3.07	3.56	3.71
min.	0.99	1.03	1.30	3.03	3.51	3.68
max.	0.99	1.05	1.42	3.21	3.63	3.78
`fast0507`, `cplex_default_transfer`, `fixed_iter` : 880.61 sec. (sequential)						
ave.	0.87	1.11	0.61	2.36	3.39	3.15
min.	0.76	1.10	0.61	2.30	3.37	3.13
max.	0.96	1.12	0.61	2.52	3.43	3.15
`fast0507`, `cplex_default_transfer`, `fixed_time` : 880.61 sec. (sequential)						
ave.	1.23	1.15	1.36	1.56	2.41	2.71
min.	1.23	1.04	1.31	1.51	2.11	2.65
max.	1.24	1.36	1.44	1.72	2.92	2.75
`mas74`, `best_bound_transfer`, `fixed_iter` : 1292.99 sec. (sequential)						
ave.	0.04	0.06	0.28	1.39	1.84	0.50
min.	0.03	0.05	0.21	1.29	1.19	0.34
max.	0.06	0.06	0.33	1.64	4.36	0.64
`mas74`, `best_bound_transfer`, `fixed_time` : 1292.99 sec. (sequential)						
ave.	0.12	0.04	0.25	1.40	3.21	0.35
min.	0.11	0.04	0.24	1.32	1.59	0.30
max.	0.12	0.04	0.27	1.45	5.40	0.39
`mas74`, `cplex_default_transfer`, `fixed_iter` : 1292.99 sec. (sequential)						
ave.	0.57	0.51	1.09	1.38	5.21	8.36
min.	0.46	0.34	0.84	1.11	4.43	7.14
max.	0.67	0.86	1.38	2.26	6.32	10.77
`mas74`, `cplex_default_transfer`, `fixed_time` : 1292.99 sec. (sequential)						
ave.	0.82	0.91	1.34	1.90	4.66	10.01
min.	0.80	0.78	1.22	1.79	4.39	8.62
max.	0.85	1.09	1.49	2.08	5.11	13.98

- `a1c1s1` : 142960.36 (sec.) with `CPX_MIPEMPHASIS_BESTBOUND`
- `arki001` : 3924.87 (sec.) with `CPX_MIPEMPHASIS_OPTIMALITY`
- `glass4` : 2001.50 (sec.) with `CPX_MIPEMPHASIS_BALANCED`
- `roll3000` : 173.73 (sec.) with `CPX_MIPEMPHASIS_BESTBOUND`
- `atlanta-ip` : 2512451.70 (sec.) with `CPX_MIPEMPHASIS_BALANCED`

These problem instances have been solved to optimality recently [5,7], and they are still hard to be solved with sequential and default solver strategies. Actually, the `roll3000` instances cannot be solved sequentially within 230464.65 sec. with `CPX_MIPEMPHASIS_BESTBOUND`. 2213435 nodes remain unexplored. Hence,

Table 2. Parallel speedups for easy instances (`cplex_default_transfer`, `fixed_time`)

	# of PEs					
	2	3	5	10	20	30
`fast0507`, `CPX_MIPEMPHASIS_BALANCED` : 880.61 sec. (sequential)						
ave.	1.23	1.15	1.36	1.56	2.41	2.71
min.	1.23	1.04	1.31	1.51	2.11	2.65
max.	1.24	1.36	1.44	1.72	2.92	2.75
`fast0507`, `CPX_MIPEMPHASIS_OPTIMALITY` : 10704.98 sec. (sequential)						
ave.	9.70	14.19	14.93	15.53	15.52	15.53
min.	9.63	13.73	14.71	15.50	15.49	15.50
max.	9.77	14.65	15.15	15.56	15.53	15.56
`fast0507`, `CPX_MIPEMPHASIS_BESTBOUND` : 15860.45 sec. (sequential)						
ave.	0.47	0.63	0.72	0.71	2.03	2.48
min.	0.44	0.48	0.72	0.68	2.01	2.45
max.	0.65	0.69	0.73	0.74	2.07	2.52
`mas74`, `CPX_MIPEMPHASIS_BALANCED` : 1292.99 sec. (sequential)						
ave.	0.82	0.91	1.34	1.90	4.66	10.01
min.	0.80	0.78	1.22	1.79	4.39	8.62
max.	0.85	1.09	1.49	2.08	5.11	13.98
`mas74`, `CPX_MIPEMPHASIS_OPTIMALITY` : 1759.18 sec. (sequential)						
ave.	1.48	1.52	1.47	2.14	7.48	11.23
min.	1.40	1.46	1.41	1.91	6.86	10.07
max.	1.56	1.64	1.62	2.27	7.86	13.09
`mas74`, `CPX_MIPEMPHASIS_BESTBOUND` : 3674.92 sec. (sequential)						
ave.	1.02	0.96	0.94	1.02	1.01	1.04
min.	1.01	0.95	0.81	0.97	0.97	0.91
max.	1.03	0.98	1.04	1.06	1.05	1.21
`harp2`, `CPX_MIPEMPHASIS_BALANCED` : 5586.38 sec. (sequential)						
ave.	8.93	5.15	8.28	8.98	13.96	24.94
min.	5.34	3.24	5.90	3.15	8.58	17.67
max.	13.70	22.00	13.55	35.23	24.76	40.86
`harp2`, `CPX_MIPEMPHASIS_OPTIMALITY` : 2014.80 sec. (sequential)						
ave.	2.16	1.87	2.53	2.47	4.15	24.16
min.	1.53	1.10	2.05	1.61	2.49	20.61
max.	3.34	2.71	4.34	6.16	7.34	32.02
`harp2`, `CPX_MIPEMPHASIS_BESTBOUND` : 2974.34 sec. (sequential)						
ave.	1.43	2.62	3.10	2.71	4.70	8.22
min.	1.24	1.88	2.12	1.78	3.11	4.03
max.	1.59	3.41	6.95	5.67	8.17	18.64
`noswot`, `CPX_MIPEMPHASIS_OPTIMALITY` : 38075.95 sec. (sequential)						
	–	–	–	27.01	119.61	276.03
`noswot`, `CPX_MIPEMPHASIS_BESTBOUND` : 193373.54 sec. (sequential)						
	–	–	–	10.58	17.44	71.02

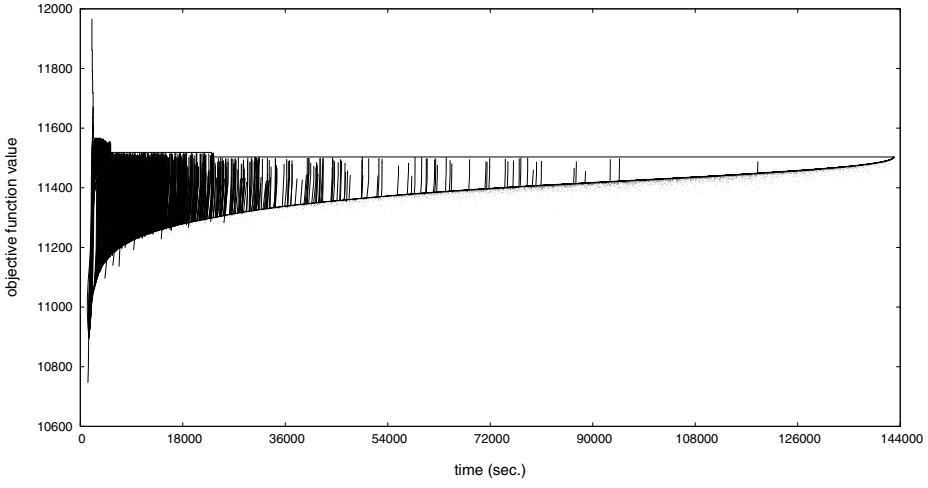


Fig. 3. Upper and lower bounds for the `a1c1s1` instance

we have a superlinear speedup result for this problem instance. We remark that R. Miyashiro reports that the `roll3000` instance were solved in about half a day with `CPX_MIPEMPHASIS_BESTBOUND` (see the MIPLIB2003 website). However, the computing environment is different: He used a 32-bit PC while we used a 64-bit machine, and the mipgap tolerances he used are different from ours (private communication).

In Figure 3, we draw upper and lower bounds as functions of time for the `a1c1s1` instance.

4 Concluding Remarks

The development of ParaLEX is still in its preliminary stages. In the initial design of ParaLEX, we intended to transfer nodes to the SubtreeSolvers only from the MainSolver or via the MainSolver to simplify the communication mechanism. However, this did not work well in many cases. Hence, we modified the transfer sequence as described in this paper. It is still a simple concept, but its implementation becomes complicated. We aim to reconsider the mechanism using what was learned in this development.

ParaLEX was originally developed by using CPLEX 9.0, but it could be compiled with CPLEX 10.1 without any modification of the codes related to the CPLEX Callable Library. Using CPLEX 9.0, ParaLEX solved the `rout` instance about 66 times faster than the `cplex` command of the version did. However, CPLEX 10.1 can solve the instance about 22 times faster than CPLEX 9.0. We observed that ParaLEX with CPLEX 10.1 is not so attractive for the `rout` instance. To solve MIP problems faster, algorithmic improvements may be more significant than that by using parallelization. On the other hand, it is still quite hard to find a good feasible solution for some problem instances. If fast finding

of a good feasible solution becomes possible by using new heuristic algorithms or different search strategies, this could lead to a tremendous speedups.

However, ParaLEX is still very attractive to solve much harder instances for CPLEX 10.1. The significance of our approach is that ParaLEX has a potential to accelerate the latest version of CPLEX using parallelization. The parallel search itself is a method to obtain a good feasible solution faster than the sequential search. This is a reason why ParaLEX achieved superlinear speedups for several problem instances. Moreover, by only using the latest version with parallelization (note that we did not use any special structure of problem instances), several of today's hardest instances of MIPLIB were solved to optimality. Therefore, ParaLEX is highly effective to solve hard problem instances.

References

1. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Oper. Res. Lett. 34, 361–372 (2006)
2. Atamtürk, A., Martin, W.P., Savelsbergh, M.W.P.: Integer-Programming Software Systems. Ann. Oper. Res. 140, 67–124 (2005)
3. Bader, D.A., Hart, W.E., Phillips, C.A.: Parallel Algorithm Design for Branch and Bound. In: Greenberg, H.J. (ed.) Tutorials on Emerging Methodologies and Applications in Operations Research, ch. 5, Kluwer Academic Press, Dordrecht (2004)
4. Crainic, T., Le Cun, B., Roucairol, C.: Parallel Branch-and-Bound Algorithms. In: Talbi, E. (ed.) Parallel Combinatorial Optimization, ch. 1, Wiley, Chichester (2006)
5. Ferris, M.: GAMS: Condor and the grid: Solving hard optimization problems in parallel. Industrial and Systems Engineering, Lehigh University (2006)
6. ILOG CPLEX 10.1 User's Manual, ILOG, Inc. (2006)
7. Laundy, R., Perregaard, M., Tavares, G., Tipi, H., Vazacopoulos, A.: Solving Hard Mixed Integer Programming Problems with Xpress-MP: A MIPLIB 2003 Case Study. Rutcor Research Report 2-2007, Rutgers University (2007)
8. Nemhauser, G.L., Wolsey, L.A.: Integer Programming and Combinatorial Optimization. John Wiley & Sons, New York (1988)
9. Ralphs, T.K.: Parallel Branch and Cut. In: Talbi, E. (ed.) Parallel Combinatorial Optimization, ch. 3, Wiley, Chichester (2006)
10. Shinano, Y., Fujie, T., Kounoike, Y.: Effectiveness of Parallelizing the ILOG-CPLEX Mixed Integer Optimizer in the PUBB2 Framework. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 451–460. Springer, Heidelberg (2003)
11. Wolsey, L.A.: Integer Programming. John Wiley & Sons, New York (1998)

Performance Analysis and Tuning of the XNS CFD Solver on Blue Gene/L

Brian J. N. Wylie[1], Markus Geimer[1], Mike Nicolai[2], and Markus Probst[2]

[1] John von Neumann Institute for Computing (NIC), Forschungszentrum Jülich GmbH, D-52425 Jülich, Germany
{b.wylie,m.geimer}@fz-juelich.de
[2] Chair for Computational Analysis of Technical Systems (CATS), Centre for Computational Engineering Science (CCES), RWTH Aachen University, D-52074 Aachen, Germany
{nicolai,probst}@cats.rwth-aachen.de

Abstract. The XNS computational fluid dynamics code was successfully running on Blue Gene/L, however, its scalability was unsatisfactory until the first Jülich Blue Gene/L Scaling Workshop provided an opportunity for the application developers and performance analysts to start working together. Investigation of solver performance pin-pointed a communication bottleneck that appeared with approximately 900 processes, and subsequent remediation allowed the application to continue scaling with a four-fold simulation performance improvement at 4,096 processes. This experience also validated the SCALASCA performance analysis toolset, when working with a complex application at large scale, and helped direct the development of more comprehensive analyses. Performance properties have now been incorporated to automatically quantify point-to-point synchronisation time and wait states in scan operations, both of which were significant for XNS on Blue Gene/L.

Keywords: performance analyses, scalability, application tuning.

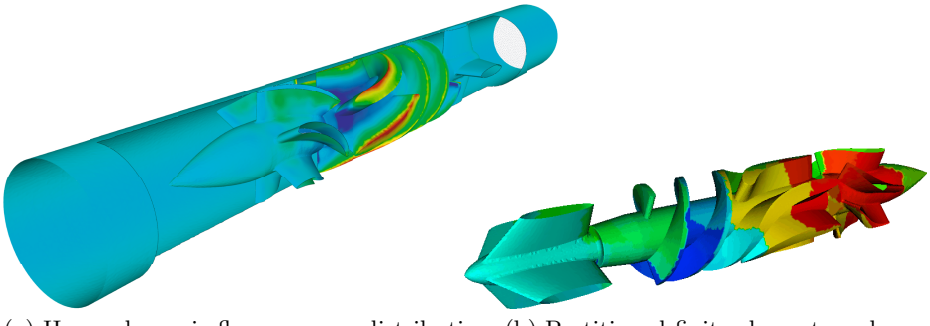
1 Introduction

XNS is an academic computational fluid dynamics (CFD) code for effective simulations of unsteady fluid flows, including micro-structured liquids, in situations involving significant deformations of the computational domain. Simulations are based on finite-element techniques using stabilised formulations, unstructured three-dimensional meshes and iterative solution strategies [1]. Main and novel areas of XNS are: simulation of flows in the presence of rapidly translating or rotating boundaries, using the shear-slip mesh update method (SSMUM); simulation of flows of micro-structured (in particular viscoelastic) liquids; and simulation of free-surface flows, using a space-time discretisation and staggered elevation-deformation-flow (EDF) approach. The parallel implementation is based on message-passing communication libraries, exploits mesh-partitioning techniques, and is portable across a wide range of computer architectures.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 107–116, 2007.

The XNS code, consisting of more than 32,000 lines of Fortran90 in 66 files, uses the EWD substrate library which fully encapsulates the use of BLAS and communication libraries, which is another 12,000 lines of mixed Fortran and C within 39 files. Although the MPI version of XNS was already ported and running on Blue Gene/L, scalability at that point was only acceptable up to 900 processes.

During early December of 2006, John von Neumann Institute for Computing (NIC) hosted the first Jülich Blue Gene/L Scaling Workshop [2], providing selected applicants an opportunity to scale their codes on the full Jülicher Blue Gene/L system (JUBL) with local NIC, IBM and Blue Gene Consortium support. JUBL is configured with 8,192 dual-core 700MHz PowerPC 440 compute nodes (each with 512MB of memory), 288 I/O nodes, and additional service and login nodes. A pair of the XNS application developers were thereby teamed with local performance analysts to investigate and resolve the application's scalability bottlenecks using the analysis tools available on the system.

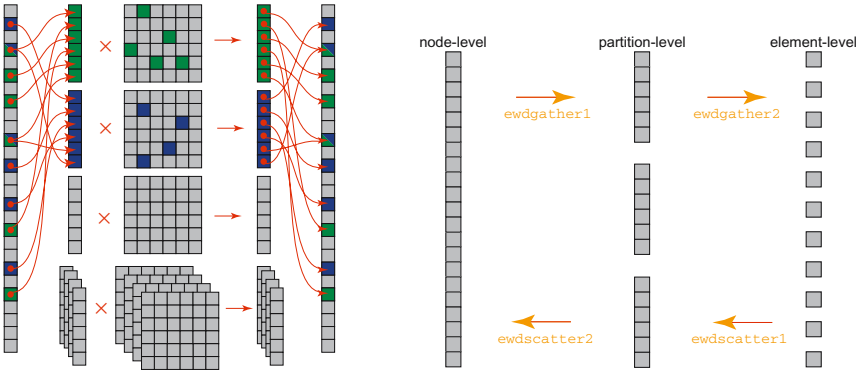


(a) Haemodynamic flow pressure distribution. (b) Partitioned finite-element mesh.

Fig. 1. DeBakey axial ventricular assist blood pump simulated with XNS

Performance on Blue Gene/L was studied with a test-case consisting of a 3-dimensional space-time simulation of the MicroMed DeBakey axial ventricular assist blood pump (shown in Figure 1). Very high resolution simulation is required to accurately predict shear-stress levels and flow stagnation areas in an unsteady flow in such a complex geometry. The mesh for the pump consisted of 3,714,611 elements (connecting 1,261,386 nodes) which were divided by the METIS graph partitioner into element sets which form contiguous subdomains that are assigned to processes.

With each set of elements assigned to a single process, the nodes are then distributed in such a way that most nodes which are interior to a subdomain are assigned to the process which holds elements of the same subdomain. Nodes at a subdomain boundary are assigned to all processes sharing that boundary. The formation of element-level components of the system of equations proceeds fully in parallel, with all data related to a given element residing in the same process. Solution of that system of equations takes place within a GMRES iterative solver, and it is here that the bulk of inter-process communication occurs, with



(a) Distributed vector-matrix multiplication. (b) Interface between different levels.

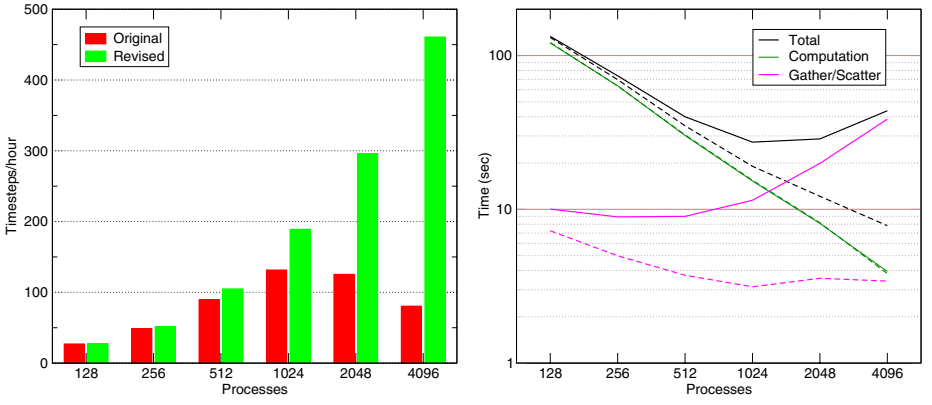
Fig. 2. Distributed sparse vector-matrix multiplications of global state vectors and partitioned matrices require data transfers taking the form of *scatter* and *gather* between node, partition and element levels

the element-based structures (stiffness matrices and local residuals) interacting with node-based structures (global residuals and increments). Figure 2(a) shows the required movement of data from element-level to node-level taking the form of a *scatter* and the reverse movement from node-level to element-level taking the form of a *gather*. These operations have two stages (Figure 2(b)): one local to the subdomain (and free of communication) and another at the surface of the subdomains (where communication is required). Four Newton-Raphson iterations are typically carried out in the solver per simulation timestep.

Time-consuming initialisation and data checkpointing (which are also highly variable due to file I/O) are excluded from measured performance reported by XNS in simulation time-steps per hour, originally peaking at around 130 timesteps/hour (Figure 3(a)). From comparison of simulation rates (and later analyses) for various numbers of timesteps, it could be determined that the first timestep's performance was representative of that of larger numbers of timesteps, allowing analysis to concentrate on simulations consisting of a single timestep.

2 XNS Execution Analysis

In addition to internal timing and reporting of the simulation timestep rate, as charted in Figure 3(a), the XNS code includes a breakdown of the performance of its primary components, namely formation of matrix left and right hand sides, GMRES solver, matrix-vector product, gather and scatter operations, etc. From a graph of these reported component costs, summarised in Figure 3(b), it was clear that the primarily computational components scaled well to larger numbers of processes, however, the gather and scatter operations used to transfer values between the node, partition and element levels became increasingly expensive. This behaviour is common to distributed-memory parallelisations of fixed-size problems, where the computational work per partition diminishes while the cost



(a) Overall simulation performance. (b) Breakdown by solver component.

Fig. 3. Comparison of original and subsequently revised XNS solver performance with DeBakey axial pump on various partition sizes of JUBL Blue Gene/L. (a) Originally unacceptable large-scale performance is improved significantly to perform over 460 timesteps/hour. (b) Breakdown of the solver component costs/timestep shows good scalability of the primarily computational components and significant improvement to Gather/Scatter scalability (original: *solid lines*, revised: *dashed lines*).

of exchanging data across partition boundaries grows with increasing numbers of processors. Understanding and optimising the communication, particularly at large processor counts, is therefore essential for effective scaling.

2.1 Profile Generation and Analysis

Several MPI profiling tools were available on JUBL, working in similar fashion and providing broadly equivalent analyses. [3,4] For example, after re-linking the XNS code with an instrumented library implementing the standard PMPI profiling interface, MPI operation characteristics were accumulated for each process during a run of the instrumented XNS executable, and these profiles were collated and presented in an analysis report upon execution completion.

From such analyses of XNS, the total time in MPI communication could be seen growing to ultimately dominate simulations, and the bulk of this time was due to rapidly growing numbers of `MPI_Sendrecv` operations within the core simulation timestep/iteration loop `ewdgather1i` and `ewdscatter2i` routines. (`MPI_Sendrecv` provides an optimised combination of `MPI_Send` and `MPI_Recv`.)

Closer examination of the profile summaries showed that each process rank makes the same number of `MPI_Sendrecv` calls in these functions, and the associated times are also very similar. Furthermore, message sizes vary considerably, with some messages of zero bytes: i.e., without message data communication. Since these operations are employed to exchange boundary elements between partitions, they could be expected to vary from process to process, however, it appeared that an exchange was done for every possible combination.

Although a zero-sized point-to-point message transfer can be useful for loose pairwise synchronisation (or coordination), it can also indicate unnecessary message traffic when there is no actual data to transfer. Unfortunately, the available profiling tools were unable to determine what proportion of `MPI_Sendrecv` operations consisted of zero-sized messages and their associated cost. This investigation could be pursued, however, via traces of the communication routines.

In addition, the profiles showed that a considerable amount of time was spent in calls to `MPI_Scan`, however, to determine whether this prefix reduction operation incurs any wait states would also require more elaborate trace analysis.

2.2 Trace Collection and Analysis

Several trace collection libraries and analysis tools were also available on JUBL, generally exclusively for tracing MPI operations. An early release (v0.5) of the SCALASCA toolset [5] had been installed for the workshop, offering tracing of MPI operations, application functions and user-specified annotations. Of particular note, execution traces from each process are unified and analysed in parallel, following the measurement and using the same computer partition. Although it had already demonstrated scalable trace collection and analysis of short benchmarks, this was an opportunity to apply it to a complex application code.

The sheer number of MPI communication operations employed by XNS each timestep was itself a significant test of SCALASCA, quickly filling trace buffers during measurement and requiring efficient internal event management during analysis/replay. Trace measurement was therefore reduced to a single simulation timestep, and analysis similarly focused to avoid the uninteresting initialisation phase (which includes file I/O that is highly variable from run to run).

Initial SCALASCA tracing simply involved re-linking the XNS code with a measurement tracing library. In this configuration, without additional instrumentation of user functions/regions, only MPI operations were traced and subsequently analysed. Traces that can be completely stored in memory avoid highly perturbative trace buffer flushing to file during measurement, and specification of appropriately-sized trace buffers was facilitated by the memory (maximum heap) usage reported by the profiling tools: fortunately, XNS memory requirements diminish with increasing numbers of processes, allowing most of the available compute node memory to be used for trace buffers.

Automatic function instrumentation is a feature of the IBM XL compilers which SCALASCA can use to track the call-path context of MPI operations and measure the time spent in non-communication functions. Unfortunately, when all functions are instrumented, measurements are often compromised by frequent calls to small functions that have a negligible contribution on overall performance but disproportionate impact on trace size and measurement perturbation. When the entire XNS application (including the EWD library) was instrumented in this fashion, ten such routines were identified that produced more events than `MPI_Sendrecv`. These routines were then specified for exclusion from measurement, resulting in traces where 94% of traced events were MPI operations (and more than 92% were `MPI_Sendrecv`).

The resulting trace analysis revealed a rich call-tree, however, navigation and analysis were encumbered by the complexity of the key nodes, often consisting of more than twenty branches at several depths. (Only subroutine names are used to distinguish successor call-paths, such that calls from different locations within a routine are aggregated.) It was therefore helpful to incorporate user-region annotation instrumentation in the XNS `hypo` routine to distinguish initialisation, simulation timestep and solver iteration loops, and finalisation phases.

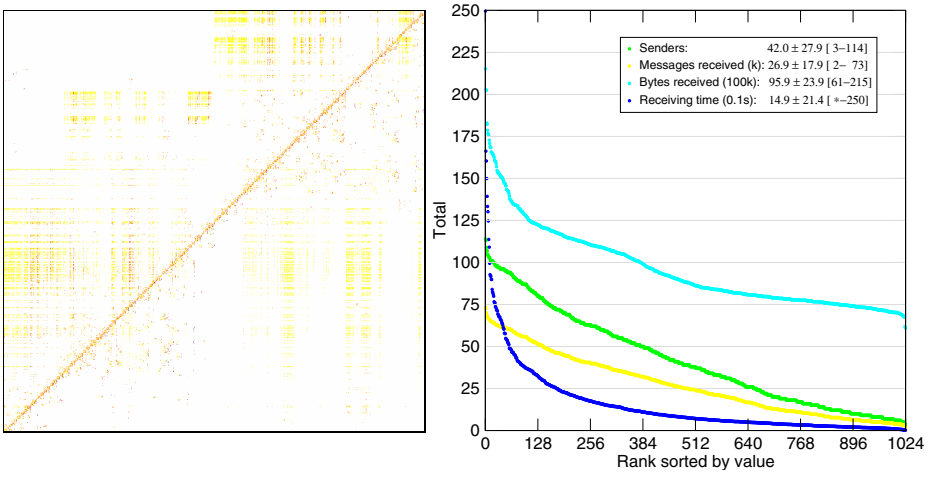
Trace measurement and analysis of the instrumented original XNS code for a single simulation timestep at a range of scales confirmed the analysis provided by the MPI profilers. With 2,048 processes, the main XNS simulation timestep loop was dilated by 15% during trace collection (compared to the uninstrumented version), producing over 23,000 million traced events which were then automatically analysed in 18 minutes.

Figure 5 (back) shows how the SCALASCA analysis report explorer highlit the most time-consuming call-paths to the `MPI_Sendrecv` operations in the `ewdscatter2` and `ewdgather1` routines of the timestep loop and presented the individual process times with the hardware topology of Blue Gene/L: MPI communication times were very balanced across processes, as evident from the 3.4% variation and uniform colouring.

As message size is logged as an attribute with each message in the trace, the number of zero-sized messages could also be determined and was found to grow rapidly with the number of processes employed (where partitions are correspondingly smaller and have fewer connections).

SCALASCA trace analysis was therefore customised to generate a report of the number of bytes received and receiving times for each sender/receiver combination and communication distribution maps were produced (e.g., Figure 4(a)). This analysis for 1,024 processes revealed that 96% of pairs had no data to exchange, and the trend makes this progressively worse for larger process configurations. Statistical analysis of the transfer data for non-zero-sized messages (Figure 4(b)) determined that *on average* each receiver rank takes 1.49 seconds to receive 9.6MB in 27,000 messages from 42 separate senders, however, there is a huge variation with maximal values typically three times the mean and 25 seconds receiving time for the rank that takes the longest.

Initial SCALASCA analyses didn't distinguish communication and synchronisation times for point-to-point messages, reporting only *Point-to-point communication time*. Incorporating a new metric for (pure) *Point-to-point synchronisation time*, for sends and receives of zero-sized messages, quantifies the very significant cost of these potentially redundant operations (Figure 5 (back)). For individual sends and receives, this was straightforward, however, the dual-nature of `MPI_Sendrecv` provides cases where only one of its send and receive parts are zero-sized and it is not possible to separate the respective costs of each part without MPI internal events [6]. After experimentation with various alternatives, it was found that only situations where the bytes sent and received are both zero could be reliably accounted as *Point-to-point synchronisation time*. Even though this underestimates the actual synchronisation cost, it ensures that the



(a) Communication time map (b) Receiver statistics

Fig. 4. Message analysis by sender/receiver for `ewdscatter` with 1,024 MPI processes. (a) The map is a matrix of message transfer times for each sender/receiver combination, coloured with a logarithmic scale: white implies no message transfer. (b) Message statistics aggregated per receiver for number of senders, messages, bytes transfered and receiving times (and then sorted by total value) highlight the considerable variation.

associated communication cost remains consistent when redundant zero-sized operations are eliminated.

Furthermore, the solver time-step loop requires around 1,500 seconds of *Collective communication time*, 53% of which is due to 11 global `MPI_Scan` operations, and almost all of it (789s) is isolated to a single `MPI_Scan` in `updateien`. Quantifying the time a scan operation on process rank n had to wait before all of its communication partners (ranks $0, \ldots, n-1$) also entered the `MPI_Scan`, another extension to the trace analysis was the implementation of a new *Early Scan* pattern. As seen in Figure 5, the aggregate *Early Scan* time is negligible for XNS, indicating that each `MPI_Scan` is called when the processes are well balanced. Further investigation of the traces determined that no rank ever exited the `MPI_Scan` before all had entered, and this generally results in longer waits for lower process ranks. Such a collective synchronisation on exit appears to be an unnecessary artifact of the MPI implementation on Blue Gene/L. While it would be desirable to define a *Scan Completion* pattern just for the cost of delayed exits from `MPI_Scan`, this requires a measure of the local scan processing time, which could only be estimated in the absence of explicit MPI internal events (e.g., via extensions to [6]). For impacted applications various remedies could be pursued: the entire MPI library or simply the implementation of `MPI_Scan` could be exchanged, or the application could try to adapt to the behaviour of the library `MPI_Scan` by redistributing or rescheduling parts of its preceding computation accordingly.

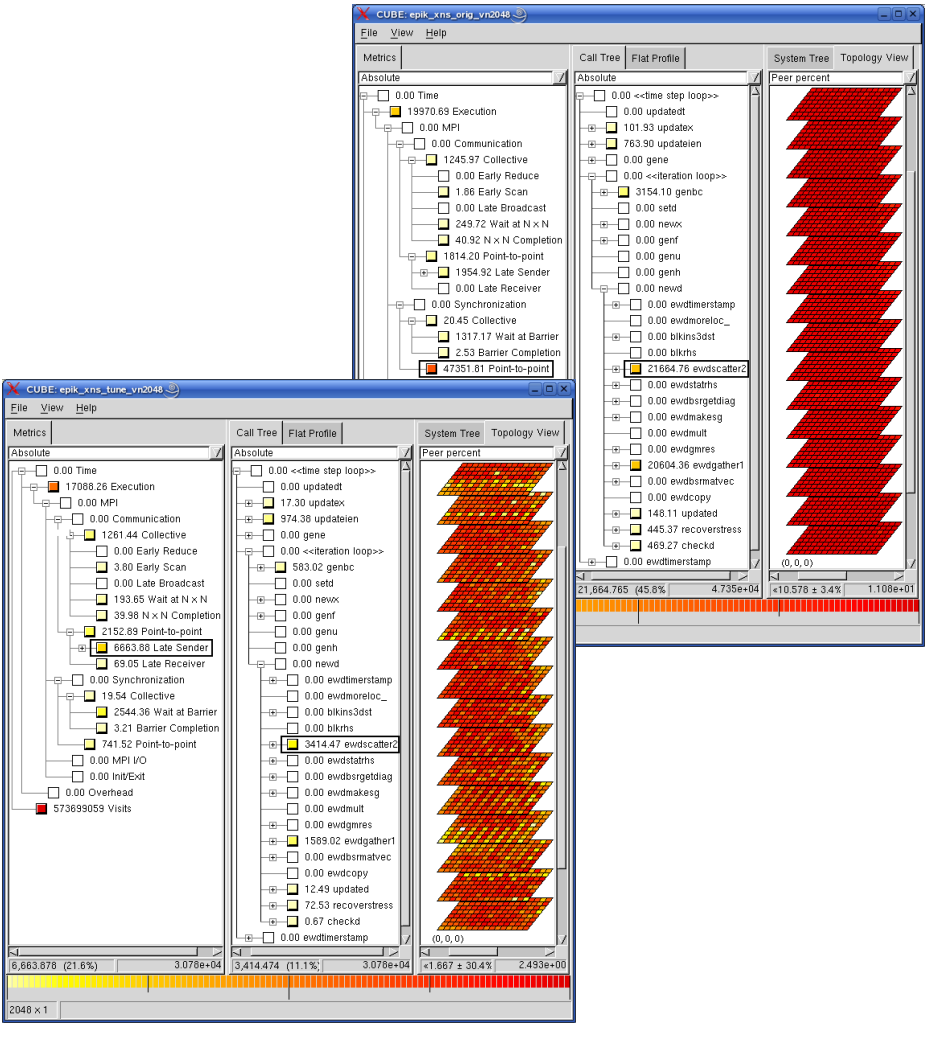


Fig. 5. SCALASCA performance analysis reports for original (*back*) and revised (*front*) versions of the XNS simulation on JUBL Blue Gene/L with 2,048 processes. A metric chosen from the metrics hierarchy (*left panes*) is shown for call-paths of the solver time-step loop (*central panes*) and its distribution per process for the selected `ewdscatter2` bottleneck (*right panes*): navigation to the most significant metric values is aided by boxes colour-coded according to each pane's specified mode and range scale (*bottom*). In each tree, collapsed nodes present inclusive metrics while expanded nodes present exclusive metrics. 47,352s of *Point-to-point synchronization time* (45.8% in `ewdscatter2` and 43.5% in `ewdgather1`) is reduced to only 742s (in routines which were not modified) by switching from `MPI_Sendrecv` to separate `MPI_Send` and `MPI_Recv` operations where zero-sized transfers are eliminated. *Point-to-point communication time*, particularly *Late Sender* situations, and *Wait at Barrier* are both significantly increased due to resulting communication load imbalance, manifest in the variation by process rank, however, overall communication time is substantially improved in the solver.

3 Modification of `ewdgather` and `ewdscatter`

The insights provided by the preceding analyses of XNS suggested splitting the `MPI_Sendrecv` operations used within the `ewdgather1` and `ewdscatter2` routines into separate `MPI_Send` and `MPI_Recv` operations which are only called when actual message data needs to be transfered. Since a static partitioning of the mesh is employed, the number of elements linking each partition is also known in advance (by potential senders and receivers), and when there are no links there is no data to transfer.

The graph comparing original and revised XNS simulation timestep rates in Figure 3(a) shows that below 1,024 processes, elimination of these zero-sized messages had little effect on the performance, which could be expected since the communication matrix remains relatively dense at this scale. Performance improved dramatically for larger configurations though, resulting in a more than four-fold overall performance improvement with 4,096 processes to over 460 timesteps/hour. Further scalability is also promising, however, lack of suitably partitioned datasets for larger numbers of processes has unfortunately prevented pursuing this investigation to date.

Elimination of zero-sized messages reduced the size of trace files collected from the new version and similarly improved trace analysis performance. Comparing analyses from the original and modified versions (Figure 5) shows the significant improvement in MPI communication time and the contributions from `ewdgather1` and `ewdscatter2`. *Point-to-point synchronisation time* decreased more than 98%, for a substantial overall performance improvement, however, the new versions of these functions show significant imbalance by process. This manifests in increased time in other parts of the solver, particularly *Wait at Barrier* and *Late Sender* situations for *Point-to-point communication* (i.e., where the receiver was blocked waiting for a sender to initiate a message transfer).

Further modifications of XNS to use asynchronous (non-blocking) message transfers within `ewdscatter2` and `ewdgather1` were investigated, but showed no additional performance improvement. This may be due to the small amount of computation available for overlap with communication within these routines. Although there is potentially more computation in the rest of the iteration loop, Figure 3(b) shows that it diminishes rapidly as the number of processes increase.

4 Conclusion

The first Jülich Blue Gene/L Scaling Workshop was a catalyst for successful collaborations between application and analysis tools developers. Analysis of the execution performance of the XNS application with more than one thousand processes was crucial in the location of adverse characteristics that developed at scale in some critical communication routines. Straightforward modification of these routines significantly improved XNS simulation performance, and enabled scaling to processor configurations four times larger than previously practical.

Further optimisations with potentially significant performance benefits are currently being evaluated, such as improved mesh partitioning and mapping

of mesh partitions onto the Blue Gene/L topology. Communication distribution maps summarising message transfers between sender/receiver combinations will be important for this purpose, and provision of these by the SCALASCA toolset is being investigated.

While the SCALASCA toolset demonstrated that it could automatically quantify and help isolate common performance problems in large-scale complex applications, various aspects could be identified for improvement. Automated trace analysis was subsequently extended to quantify inefficiencies in `MPI_Scan` and `MPI_Sendrecv`, the latter being found to be responsible for costly and unnecessary point-to-point synchronisations. Synchronisation and communication costs are currently based on heuristics that ensure analysis consistency, yet which might be determined more accurately in future.

The profiling tools available on Blue Gene/L were a convenient starting point for performance analysis, however, they provided limited insight into synchronisation costs and imbalance. Message statistics for communication and synchronisation operations can be calculated from trace analysis or accumulated during measurement, and these capabilities are now being incorporated in the SCALASCA toolset. By integrating runtime summarisation and tracing capabilities, convenience of use is being pursued particularly for measurement configuration and selective event tracing.

The open-source SCALASCA toolset is freely available for download [7].

Acknowledgements. We would like to thank the sponsors, organisers and co-participants of the Jülich Blue Gene/L Scaling Workshop for the valuable opportunity to benefit from their assistance, advice and productive atmosphere. We also thank the reviewers for their constructive suggestions.

References

1. Behr, M., Arora, D., Coronado, O., Pasquali, M.: Models and finite element techniques for blood flow simulation. Int'l J. Computational Fluid Dynamics 20, 175–181 (2006)
2. Frings, W., Hermanns, M.-A., Mohr, B., Orth, B. (eds): Jülich Blue Gene/L Scaling Workshop (December 2006) Forschungszentrum Jülich ZAM-IB-2007-01 `http://www.fz-juelich.de/zam/bgl-sws06/`
3. IBM Advanced Computing Technology Center: High Performance Computing Toolkit, `http://www.research.ibm.com/actc/`
4. Vetter, J., Chambreau, C.: MPIP — lightweight, scalable MPI profiling (2005) `http://www.llnl.gov/CASC/mpip/`
5. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)
6. Jones, T., et al.: MPI PERUSE: A performance revealing extensions interface to MPI. `http://www.mpi-peruse.org/`
7. Forschungszentrum Jülich GmbH: SCALASCA: Scalable performance analysis of large-scale parallel applications. `http://www.scalasca.org/`

(Sync|Async)$^+$ MPI Search Engines

Mauricio Marin[1] and Veronica Gil Costa[2]

[1] Yahoo! Research, Santiago, University of Chile
[2] DCC, University of San Luis, Argentina

Abstract. We propose a parallel MPI search engine that is capable of automatically switching between asynchronous message passing and bulk-synchronous message passing modes of operation. When the observed query traffic is small or moderate the standard multiple managers/workers thread based model of message passing is applied for processing the queries. However, when the query traffic increases a round-robin based approach is applied in order to prevent from unstable behavior coming from queries demanding the use of a large amount of resources in computation, communication and disk accesses. This is achieved by (**i**) a suitable object-oriented multi-threaded MPI software design and (**ii**) an "atomic" organization of the query processing which allows the use of a novel control strategy that decides the proper mode of operation.

1 Introduction

The distributed inverted file is a well-known index data structure for supporting fast searches on Search Engines dealing with very large text collections [1,2,3,4,5,7,8]. An inverted file is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of relevant terms found in the collection. Each of these terms is associated with a posting list which contains the document identifiers where the term appears in the collection along with additional data used for ranking purposes. To solve a query, it is necessary to get the set of documents associated with the query terms and then perform a ranking of these documents in order to select the top K documents as the query answer. In this paper we assume posting list items composed of pairs of document identifier and frequency in which the associated term appears in the given document.

The approach used by well-known Web Search Engines to the parallelization of inverted files is pragmatic, namely they use the document partitioned approach. Documents are evenly distributed on P processors and an independent inverted file is constructed for each of the P sets of documents. The disadvantage is that each user query has to be sent to the P processors which leads this strategy to a poor $O(P)$ scalability. Apart from the communication cost, sending a copy of every query to each processor increases overheads associated with large number of threads and disk operations that have to be scheduled. It can also present imbalance at posting lists level (this increases disk access and interprocessor communication costs). The advantage is that document partitioned indexes are

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 117–124, 2007.

easy to maintain since insertion of new documents can be done locally and this locality is extremely convenient for the posting list intersection operations required to solve the queries (they come for free in terms of communication costs).

Another competing approach is the term partitioned index in which a single inverted file is constructed from the whole text collection to then distribute evenly the terms with their respective posting lists onto the processors. However, the term partitioned inverted file destroys the possibility of computing intersections for free in terms of communication cost and thereby one is compelled to use strategies such as smart distribution of terms onto processors to increase locality for most frequent terms (which can be detrimental for overall load balance) and caching. However, it is not necessary to broadcast queries to all processors. Nevertheless, the load balance is sensitive to queries referring to particular terms with high frequency and posting lists of differing sizes. In addition index construction and maintenance is much more costly in communication. However, this strategy is able to achieve $O(1)$ scalability.

Most implementations of distributed inverted files reported so far are based on the message passing approach to parallel computing in which we can find combinations of multithreaded and computation/communication overlapped systems. The typical case is to have in each of the P processing nodes a set of threads dedicated to receive queries and communicate with other nodes (threads) in order to produce an answer in the form of the top K documents that satisfy the query. However, it is known that threads can be potential sources of overheads and can produce unpredictable outcomes in terms of running time. Still another source of unpredictable behavior can be the accesses to disk used to retrieve the posting lists. Some queries can demand the retrieval of very large lists from secondary memory involving hundreds of disk blocks.

In that context the principle behind the findings reported in this paper can be explained by analogy with the classic round-robin strategy for dealing with a set of jobs competing to receive service from a processor. Under this strategy every job is given the same quantum of CPU so that jobs requiring large amounts of processing cannot monopolize the use of the CPU. This scheme can be seen as bulk-synchronous in the sense that jobs are allowed to perform a set of operations during their quantum. In our setting we define quantums in computation, disk accesses and communication given by respective "atoms" of size K where K is the number of documents to be presented to the user. We use a relaxed form of bulk-synchronous parallel computation [9] to process those atoms in parallel in a controlled (synchronous) manner with atoms large enough to properly amortize computation, disk and communication overheads.

For instance, for a moderate query traffic $q = 32$ and using a BSP library built on top of MPI (BSPonMPI http://bsponmpi.sourceforge.net/) we found this bulk-synchronous way of query processing quite efficient with respect to message passing realizations in MPI and PVM. See figure 1 that shows results for two text collections indexed using the document (D) and term (T) partitioned indexes. Notice that the technical details of the experiments reported in this

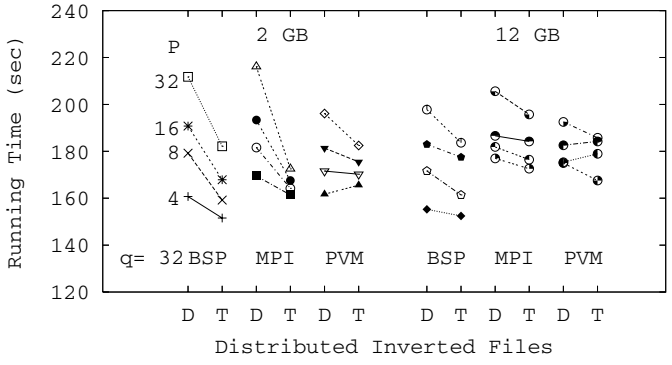


Fig. 1. Comparing BSP, MPI and PVM for inverted files under moderate query traffic

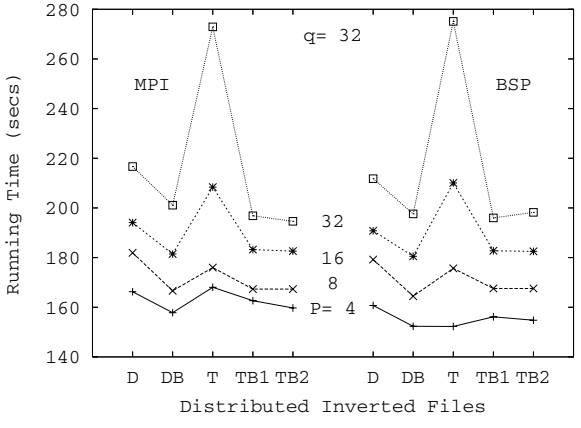


Fig. 2. Comparing BSP with a semi-synchronous MPI realization in which each processors waits to receive at least one message from all other processors before continuing query processing

paper are given in the Appendix. Also larger number of processors P implies larger running times because we inject in each processor a constant number of queries. This because the inter-processor communication cost is always an increasing function of P for any architecture. These results shows that our realizations of inverted files scale up efficiently because in each curve we duplicate the number of processors and running times increase modestly as $O(\log P)$.

However, we also observed that with a semi-synchronous MPI realization we were able to achieve similar performance to BSP. In this case we force every MPI processor to wait for P messages (one per processor) before delivering them to their target threads. The results are in figure 2 which shows other alternative implementations of inverted files where DB and TB represent bucketing strategies devised for improving load balance and T a bad (but in use) idea for

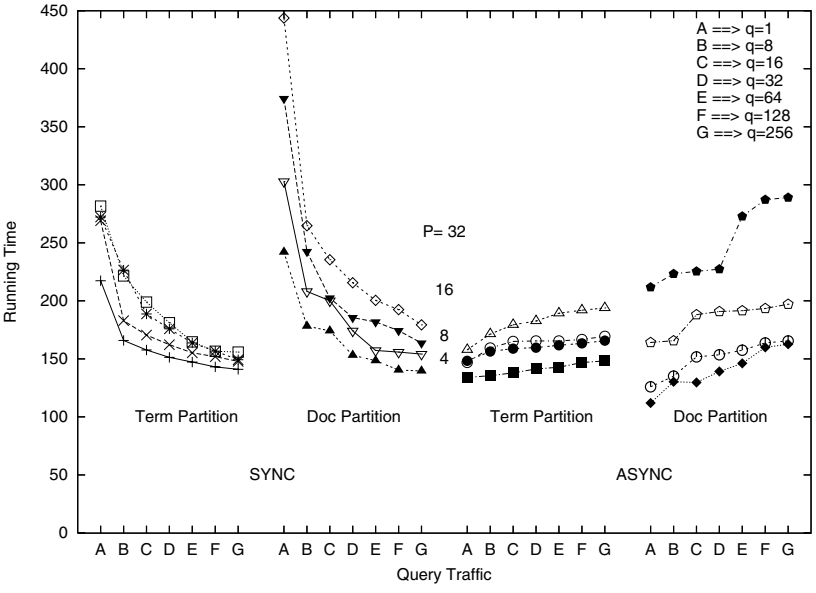


Fig. 3. Comparing MPI under bulk-synchronous (SYNC) and asynchronous (ASYNC) modes of operation for the term and document partitioned inverted files under different query traffics of q queries per processor per unit time for 32, 16, 8 and 4 processors

implementing the term partitioned index. These results are evidence that in practice for this application of parallel computing it is not necessary to barrier synchronize all of the processors, it suffices to synchronize locally at processor level from messages arriving from the other processors.

Nevertheless the situation was quite different when we considered cases of low traffic of queries. The figure 3 shows results for two MPI realizations of the inverted files, namely an asynchronous message passing multi-threaded (Async) version and a non-threaded semi-bulk-synchronous MPI (Sync) realizations (details in the next section). This clearly makes a case for a hybrid implementation which is the discussion of this paper.

In the remainder of this paper we describe our proposal to make possible this hybrid form of parallel query processing using MPI. The rule to decide between one or another mode of operation is based on the simulation of a BSP computer. This simulation is performed on-the-fly as queries are received and send to processing.

2 Round-Robin Query Processing

In this section we describe our algorithm for query processing and method for deciding between asynchronous and synchronous modes of operation. The parallel processing of queries is basically composed of a phase in which it is necessary

to fetch parts of all of the posting lists associated with each term present in the query, and perform a ranking of documents in order to produce the results. After this, additional processing is required to produce the answer to the user. At the parallel server side, queries arrive from a receptionist machine that we call the *broker*. The broker machine is in charge of routing the queries to the cluster's processors and receiving the respective answers. It decides to which processor to route a given query by using a load balancing heuristic. The particular heuristic depends on the approach used to partition the inverted file. Overall the broker tends to evenly distribute the queries on all processors.

The processor in which a given query arrives is called the *ranker* for that query since it is in this processor where the associated document ranking is performed. Every query is processed using two major steps: the first one consists on fetching a K-sized piece of every posting list involved in the query and sending them to the ranker processor. In the second step, the ranker performs the actual ranking of documents and, if necessary, it asks for additional K-sized pieces of the posting lists in order to produce the K best ranked documents that are passed to the broker as the query results. We call this *iterations*. Thus the ranking process can take one or more iterations to finish. In every iteration a new piece of K pairs (doc_id, frequency) from posting lists are sent to the ranker for every term involved in the query. In this scheme, the ranking of two or more queries can take place in parallel at different processors together with the fetching of K-sized pieces of posting lists associated with other queries.

We use the vectorial method for performing the ranking of documents along with the filtering technique proposed in [6]. Consequently, the posting lists are kept sorted by frequency in descending order. Once the ranker for a query receives all the required pieces of posting lists, they are merged into a single list and passed throughout the filters. If it happens that the document with the least frequency in one of the arrived pieces of posting lists passes the filter, then it is necessary to perform a new iteration for this term and all others in the same situation. We also provide support for performing the intersection of posting lists for boolean AND queries. In this case the ranking is performed over the documents that contain all the terms present in the query.

The synchronous search engine is implemented on top of the BSP model of parallel computing [9] as follows. In BSP the computation is organized as a sequence of *supersteps*. During a superstep, the processors may perform computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors. The underlying communication library ensures that all messages are available at their destinations before starting the next superstep.

Thus at the beginning of each superstep the processors get into their input message queues both new queries placed there by the broker and messages with pieces of posting lists related to the processing of queries which arrived at previous supersteps. The processing of a given query can take two or more supersteps to be completed. All messages are sent at the end of every superstep and thereby

they are sent to their destinations packed into one message per destination to reduce communication overheads.

Query processing is divided in "atoms" of size K, where K is the number of documents presented to the user as part of the query answer. These atoms are scheduled in a round-robin manner across supersteps and processors. The asynchronous tasks are given K sized quantums of processor time, communication network and disk accesses. These quantums are granted during supersteps, namely they are processed in a bulk-synchronous manner. As all atoms are equally sized then the net effect is that no particular task can restrain others from using the resources. During query processing, under an observed query traffic of $Q = q\,P$ queries per unit time with q per processor per superstep, the round-robin principle is applied as follows. Once Q new queries are evenly injected onto the P processors, their processing is started in iterations as described above. At the end of the next superstep some queries, say n queries, all requiring a single iteration, will finish their processing and thereby at the following superstep n new queries can start their processing. Queries requiring more iterations will continue consuming resources during a few more supersteps.

In each processor we maintain several threads which are in charge of processing the K-sized atoms. We use LAM-MPI so we put one thread to perform the interprocessors message passing. This thread acts as a scheduler and message router for the main threads in charge of solving the queries. We use the non-blocking message passing communication primitives. We organized our C++ code into a set of objects, among them we have the object called "processor" which is the entry point to all the index methods and ranking. The crucial point here is that all threads have access to this object and concurrency conflicts are avoided by keeping in thread's local memory the context of each queries they are in charge of. When the search engine switches to bulk-synchronous operation all threads are put to sleep on condition variables and the main thread takes control of processing sequentially the different stages of queries during supersteps.

When the search engine is operating in the asynchronous mode it simulates the operation of a BSP machine. This is effected every N_q completed queries per processor as follows. During the interval of N_q queries each processor of the asynchronous machine registers the total number of iterations required by each query being solved. The simulation of a BSP computer for the same period can be made by assuming that q new queries are received in each superstep and processor. For this period of Δ units of time, the observed value of q can be estimated in a very precise manner by using the G/G/∞ queuing model. Let S be the sum of the differences [DepartureTime - ArrivalTime] of queries, that is the sum of the intervals of time elapsed between the arrival of the queries and the end of their complete processing. Then the average q for the period is given by S/Δ. This because the number of active servers in a G/G/∞ system is defined as the ratio of the arrival rate of events to the service rate of events (λ/μ). If n queries are received by the processor during the interval Δ, then the arrival rate is $\lambda = n/\Delta$ and the service rate is $\mu = n/S$.

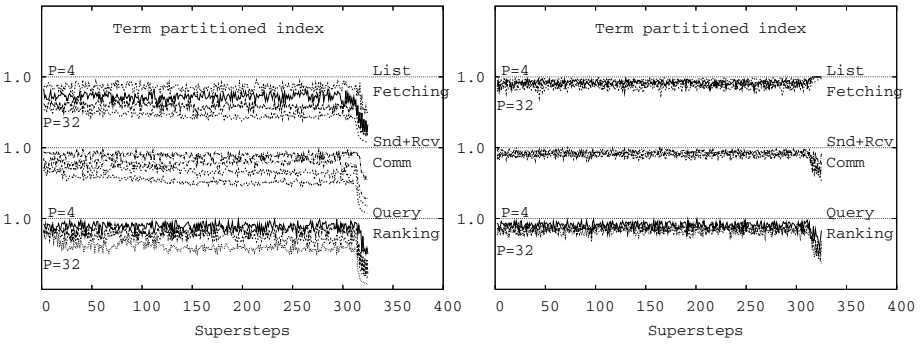


Fig. 4. Predicted SYNC efficiencies in disk accesses, communication and query ranking. Figure [left] is a case in which the query traffic is very low ($q = 1$) and figure [right] is a case of high traffic ($q = 128$). These extreme cases explains the performance of the SYNC term partitioned index in figure 3.

In addition the processors maintain the number of "atoms" of each type processed during the interval of running time. The efficiency metric is used to determine when to switch from one mode of operation to the another. For a metric x this is defined as the ration average(x)/maximum(x) both values taken over all processors and averaging across supersteps. The search engine switches to bulk-synchronous mode when efficiencies in ranking, communication and list-fetching are over 80%. Below that the asynchronous message passing mode is used. We have found that this simulation is accurate as a predictor of performance. For instance, this simulation predicts the efficiencies shown in figure 4 which are consequent with the bad and good performances observed in figure 3 for the term partitioned index for the same experiments in both cases.

3 Conclusions

We have presented a method and a MPI-based implementation to allow a search engine to dynamically switch its mode of parallel processing between asynchronous and bulk-synchronous message passing. This is achieved by dividing the tasks involved in the processing of queries into K-sized single-units and interleaving their execution across processors, network communication and disk-accesses. The glue between the two modes of operation is the simulation of a bulk-synchronous parallel computer. Our experiments show that this simulation is quite accurate and independent of the actual mode of operation of the search engine, be it under low or high traffic of queries.

References

1. Badue, C., Baeza-Yates, R., Ribeiro, B., Ziviani, N.: Distributed query processing using partitioned inverted files. In: Eighth Symposium on String Processing and Information Retrieval (SPIRE'01), pp. 10–20 (2001)

2. Cacheda, F., Plachouras, V., Ounis, I.: Performance analysis of distributed architectures to index one terabyte of text. In: McDonald, S., Tait, J. (eds.) Proc. ECIR European Conf. on IR Research, Sunderland, UK, pp. 395–408 (2004)
3. Jeong, B.S., Omiecinski, E.: Inverted file partitioning schemes in multiple disk systems. IEEE Transactions on Parallel and Distributed Systems 16, 142–153 (1995)
4. MacFarlane, A.A., McCann, J.A., Robertson, S.E.: Parallel search using partitioned inverted files. In: 7th International Symposium on String Processing and Information Retrieval, pp. 209–220. IEEE CS Press, Los Alamitos (2000)
5. Moffat, W., Webber, J., Zobel, B.-Y.R.: A pipelined architecture for distributed text query evaluation. Information Retrieval (2006)
6. Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrival with frequency-sorted indexes. Journal of the American Society for Information Science 47(10), 749–764 (1996)
7. Ribeiro-Neto, B.A., Barbosa, R.A.: Query performance for tightly coupled distributed digital libraries. In: Third ACM Conference on Digital Libraries, pp. 182–190. ACM Press, New York (1998)
8. Stanfill, C.: Partitioned posting files: a parallel inverted file structure for information retrieval. In: 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Brussels, Belgium, pp. 413–428. ACM Press, New York (1990)
9. Valiant, L.: A bridging model for parallel computation. Comm. ACM 33, 103–111 (1990)

Appendix

We use two text databases, a 2GB and 12GB samples of the Chilean Web taken from the `www.todocl.cl` search engine. The text is in Spanish. Using this collection we generated a 1.5GB index structure with 1,408,447 terms. Queries were selected at random from a set of 127,000 queries taken from the todocl log. The experiments were performed on a cluster with dual processors (2.8 GHz) that use NFS mounted directories. In every run we process 10,000 queries in *each* processor. That is the total number of queries processed in each experiment reported in this paper is 10,000 P. For our collection the values of the filters C_{ins} and C_{add} of the filtering method described in [6] were both set to 0.1 and we set K to 1020. On average, the processing of every query finished with $0.6K$ results after 1.5 iterations. Before measuring running times and to avoid any interference with the file system, we load into main memory all the files associated with queries and the inverted file.

A Case for Standard Non-blocking Collective Operations

Torsten Hoefler[1,4], Prabhanjan Kambadur[1], Richard L. Graham[2], Galen Shipman[3], and Andrew Lumsdaine[1]

[1] Open Systems Lab, Indiana University, Bloomington IN 47405, USA, {htor,pkambadu,lums}@cs.indiana.edu
[2] National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge TN 37831, USA, rlgraham@ornl.gov
[3] Advanced Computing Laboratory, Los Alamos National Laboratory, Los Alamos, NM 87545, USA, LA-UR-07-3159 gshipman@lanl.gov
[4] Chemnitz University of Technology, 09107 Chemnitz, Germany, htor@cs.tu-chemnitz.de

Abstract. In this paper we make the case for adding standard non-blocking collective operations to the MPI standard. The non-blocking point-to-point and blocking collective operations currently defined by MPI provide important performance and abstraction benefits. To allow these benefits to be simultaneously realized, we present an application programming interface for non-blocking collective operations in MPI. Microbenchmark and application-based performance results demonstrate that non-blocking collective operations offer not only improved convenience, but improved performance as well, when compared to manual use of threads with blocking collectives.

1 Introduction

Although non-blocking collective operations are notably absent from the MPI standard, recent work has shown that such operations can be beneficial, both in terms of performance and abstraction. Non-blocking operations allow communication and computation to be overlapped and thus to leverage hardware parallelism for the asynchronous (and/or network-offloaded) message transmission. Several studies have shown that the performance of parallel applications can be significantly enhanced with overlapping techniques (e.g., cf. [1,2]). Similarly, collective operations offer a high-level interface to the user, insulating the user from implementation details and giving MPI implementers the freedom to optimize their implementations for specific architectures.

In this paper, we advocate for standardizing non-blocking collective operations. As with non-blocking point-to-point operations and blocking collective operations, the performance and abstraction benefits of non-blocking collective

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 125–134, 2007.

operations can only be realized if these operations are represented with a procedural abstraction (i.e., with an API). Although a portable library on top of MPI could be used to provide non-blocking collective functionality to the HPC community, standardization of these operations is essential to enabling their widespread adoption. In general, vendors will only tune operations that are in the standard and users will only use features that are in the standard.

It has been suggested that non-blocking collective functionality is not needed explicitly as part of MPI because a threaded MPI library could be used with collective communication taking place in a separate thread. However, there are several drawbacks to this approach. First, it requires language and operating system support for spawning and managing threads, which is not possible on some operating systems—in particular on operating systems such as Catamount designed for HPC systems. Second, programmers must then explicitly manage thread and synchronization issues for purposes of communication even though these issues could and should be hidden from them (e.g., handled by an MPI library). Third, the required presence of threads and the corresponding synchronization mechanisms imposes the higher cost of thread-safety on all communication operations, whether overlap is obtained or not (cf. [3]). Finally, this approach provides an asymmetric treatment of collective communications with respect to point-to-point communications (which do support asynchronous communications).

Non-blocking collective operations provide some performance benefits that may only be seen at scale. The scalability of large scientific application codes is often dependent on the scalability of the collective operations used. At large scale, system noise affects the performance of collective communications more than it affects the performance of point-to-point operations. because of the ordered communications patterns use by collective communications algorithms. To continue to scale the size of HPC systems to peta-scale and above, we need communication paradigms that will admit effective use of the hardware resources available on modern HPC systems. Implementing collective operations so that they do not depend on the the main CPU is one important means of reducing the effects of system noise on application scalability.

1.1 Related Work

Several efforts have studied the benefits of overlapping computation with communications, with mixed results. Some studies have shown that non-blocking collective operations improve performance, and in other cases a bit of performance degradation was observed. Danalis et.al. [2] obtained performance improvement by replacing calls to MPI blocking collectives with calls to non-blocking MPI point-to-point operations. Kale et al. [4] analyzed the applicability of a non-blocking personalized exchange to a small set of applications. Studies such as [1,5] mention that non-blocking collective operations would be beneficial but do not quantify these benefits. IBM extended the standard MPI interface to include non-blocking collectives in their parallel environment (PE), but have dropped support for this non-standard functionality in the latest release of this PE. Due

to its channel semantics, MPI/RT [6] defines all operations, including collective operations, in a non-blocking manner. Hoefler et. al. [7,8] have shown that non-blocking collective operations can be used to improve the performance of parallel applications. Finally, several studies of the use of non-blocking collectives to optimize three-dimensional FFTs have been done [5,9,10,11]. The results of applying these non-blocking communication algorithms (replacing MPI All-To-All communications) were inconclusive. In some cases the non-blocking collectives improved performance, and in others performance degraded a bit.

The remainder of the paper is structured as follows. Section 2 describes our proposed application programming interface followed by a discussion of different implementation options in Section 3. Microbenchmarks of two fundamentally different implementations are presented in Section 4. Section 5 presents a case study of the applicability of non-blocking collective operations to the problem of a parallel three-dimensional Fourier Transformation.

2 Application Programming Interface

We propose an API for the non-blocking collectives that is very similar to that of the blocking collectives and the former proprietary IBM extension. We use a naming scheme similar to the one used for the non-blocking point-to-point API (e.g., MPI_Ibarrier instead of MPI_Barrier). In addition, request objects (MPI_Request) are used for a completion handle. The proposed interfaces to all collective operations are defined in detail in [12]. For example, a non-blocking barrier would look like:

```
1  MPI_Ibarrier(comm, request);
   ...
   /* computation, other MPI communications */
   ...
   MPI_Wait(request, status);
```

Our interface relaxes the strict MPI convention that only one collective operation can be active on any given communicator. We extend this so that we can have a huge number (system specific, indicated by MPI_ICOLL_MAX_OUTSTANDING, cf. [12]) of parallel non-blocking collectives and a single blocking collective outstanding at any given communicator. Our interface does not introduce collective tags to stay close to the traditional syntax of collective operations. The order of issuing a given collective operation defines how the collective-communications traffic matches up across communicators. Similar to point-to-point communications, progress for these non-blocking collective operations depends on both underlying system hardware and software capabilities to support asynchronous communications, as well implementation of these collectives by the MPI library. In some cases MPI_Test or MPI_Wait may need to be called to progress these non-blocking collective operations. This may be particularly true for collective operations that transform user data, such as MPI_Allreduce.

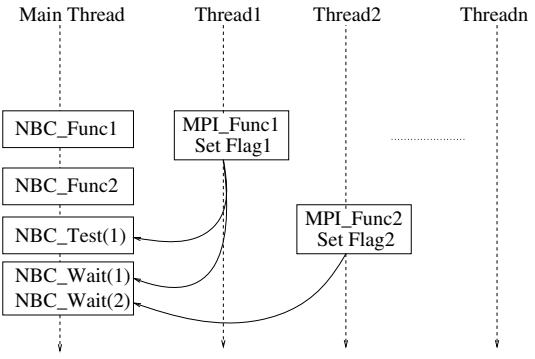


Fig. 1. Execution of NBC_ calls in separate threads

3 Advice to Implementors

There are two different ways to implement support for non-blocking collective operations. The first way is to process the blocking collective operation in a separate thread and the second way is to implement it on top of non-blocking point-to-point operations. We will evaluate both implementations in the following. We use a library approach, e.g., both variants are implemented in a library with a standardized interface which is defined in [12]. This enables us to run identical applications and benchmarks with both versions. The following section discusses our implementation based on threads.

3.1 Implementation with Threads

The threaded implementation, based on the pthread interface, is able to spawn a user-defined number of communication threads to perform blocking collective operations. It operates using a task-queue model, where every thread has its own task queue. Whenever a non-blocking collective function is called, a work packet (containing the function number and all arguments) is placed into the work queue of the next thread in a round robin scheme.

The worker threads could either poll their work queue or use condition signals to be notified. Condition signals may introduce additional latency while constant polling increases the CPU overhead. We will analyze only the condition wait method in the next section because experiments with the polling method showed that it is worse in all regards. Since there must be at least one worker thread per MPI job, at most half of the processing cores is available to compute unless the system is oversubscribed.

Whenever a worker thread finds a work packet in its queue (either during busy waiting or after being signaled), the thread starts the corresponding collective MPI operation and sets a flag after its completion. All asynchronous operations have to be started on separate communicators (mandated by the MPI standard). Thus, every communicator is duplicated on its first use with any non-blocking

collective and cached for later calls. Communicator duplication is a blocking collective operation in itself and causes matching problems when it's run with threads (cf. [3]). The communicator duplication has to be done in the user thread to avoid any race conditions, which makes the first call to a non-blocking collective operation with every communicator block. All subsequent calls are executed truly non-blocking.

When the user calls NBC_Test, the completion flag is simply checked and the appropriate return code generated. A call to NBC_Wait waits on a condition variable.

3.2 Implementation with Non-blocking Point-to-Point

The point-to-point message based implementation is available in LibNBC. LibNBC is written in ANSI C using MPI-1 functionality exclusively to ensure highest portability. The full implementation is open source and available for public download on the LibNBC website [13]. The detailed implementation documentation is provided in [14], and the most important issues are discussed in the following.

The central part of LibNBC is the collective schedule. A schedule consists of the operations that have to be performed to accomplish the collective task (e.g., an MPI_Isend, MPI_Irecv). The collective algorithm is implemented like in the blocking case, based on point-to-point messages. But instead of performing all operations immediately, they are saved in the collective schedule together with their dependencies. However, the internal interface to add new algorithms is nearly identical to the MPI interface. A detailed documentation about the addition of new collective algorithms and the internal and external programming interfaces of LibNBC is available in [14]. The current implementation is optimized for InfiniBand™ and implements different algorithms for most collective operations (cf. [15]).

All communications require an extra communicator to prevent collisions with the user program. This communicator is duplicated from the original one in the first NBC call with a new communicator and cached for subsequent calls. This makes the first call blocking, as in the threaded implementation described in the previous section.

4 Microbenchmarking the Implementations

We developed a micro-benchmark to assess the performance and overlap potential of both implementations of non-blocking collective operations. This benchmark uses the interface described in [12]. For a given collective operation, it measures the time to perform the blocking MPI collective, the non-blocking collective in a blocking way (without overlap) and the non-blocking collective interleaved with busy loops to measure the potential computation and communications overlap. A detailed description of the benchmark is available in [8]. In addition, the benchmark measures the communication overhead of blocking and

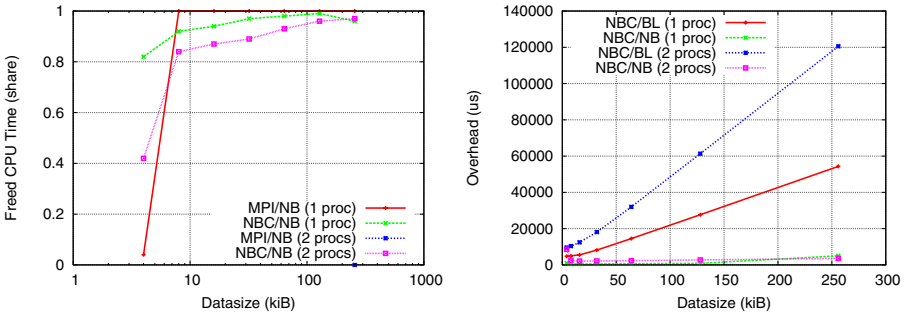


Fig. 2. Left: Share of the freed CPU time with the non-blocking MPI and NBC alltoall operation with regards to the blocking MPI implementation. Right: Blocking and non-blocking NBC_Ialltoall overhead for different CPU configurations. Measured with 128 processes in 64 and 128 nodes respectively.

non-blocking collective operations. Overhead is defined as the time the calling thread spends in communications related routines, i.e., the time the thread can't spend doing other work. The communication overhead of blocking operations is the amount of time to finish the collective operation, as the collective call does not complete until the collective operation had completed locally. Non-blocking operations allow for overlap of the communication latency and the overhead has the potential to be less than the time to complete the given collective, and providing the calling thread compute cycles. The communication overhead of non-blocking operations is highly implementation, network, and communications stack dependent. We could not run these thread-based tests on the Cray XT4, as it does not provide thread support.

Using both implementations and the benchmark results, four different times are measured:

- Blocking MPI collective in the user thread (MPI/BL)
- Blocking MPI collective in a separate communications thread to emulate non-blocking behavior (MPI/NB)
- Non-blocking NBC operation without overlap, i.e., the initiation is directly followed by a wait (NBC/BL)
- Non-blocking NBC operation with maximum overlap, i.e., computing at least as long as an NBC/BL operation takes (NBC/NB)

We benchmarked both implementations with Open MPI 1.2.1 [16] on the Coyote cluster system at Los Alamos National Labs, a 1290 node AMD Opteron cluster with an SDR InfiniBand network. Each node has two single core 2.6 GHz AMD Opteron processors, 8 GBytes of RAM and a single SDR InfiniBand HCA. The cluster is segmented into 4 separate scalable units of 258 nodes. The largest job size that can run on this cluster is therefore 516 processors.

Figure 2 shows the results of the microbenchmark for different CPU configurations of 128 processes running on Coyote. The threaded MPI implementation

allows nearly full overlap (frees nearly 100% CPU) as long as the system is not oversubscribed, i.e., every communication thread runs on a separate core. However, this implementation fails to achieve any overlap (it shows even negative impact) if all cores are used for computation. The implementation based on non-blocking point-to-point (LibNBC) allows decent overlap in all cases, even if all cores are used for computation.

5 Case Study: Three-Dimensional FFT

Parallel multi-dimensional Fast Fourier Transformations (FFTs) are used as compute kernels in many different applications, such as quantum mechanical or molecular dynamic calculations. In this paper we also study the application of non-blocking collective operations to optimize a three-dimensional FFT to demonstrate the benefit of overlapping computation with communication for this important kernel. This operation is used at least once per application computational step.

The three-dimensional FFT can be split into three one-dimensional FFTs performed along all data points. We use FFTW to perform the 1d-FFTs and distribute the data block-wise (blocks of xy-planes) so that the x and y dimensions can be transformed without redistributing the data between processors. The z transformation requires a data redistribution among all nodes which is efficiently implemented by an MPI_Alltoall function.

A pipeline scheme is used for the communication. As soon as the first data elements (i.e., planes) are ready, the communication of them is started in a non-blocking way with NBC_Ialltoall. This enables the communication of those elements to overlap with the computation of all following elements. As soon as the last element is computed and its communication is started, all outstanding collective operations are completed with NBC_Wait (i.e., the last operation has no overlap potential).

We benchmark the strong scaling of a full transformation of a 1024^3 point FFT box (960^3 for 32 processes due to memory limitations) on the the Cray XT4, Jaguar, at the National Center for Computational Sciences, Oak Ridge National Laboratory. This cluster is made up of a total of 11,508 dual socket 2.6 GHz dual-core AMD Opteron chips, and the network is a 3-D torus with the Cray-designed SeaStar [17] communication processor and network router designed to offload network communication from the main processor. The compute nodes run the Catamount lightweight micro-kernel. All communications use the Portals 3.3 communications interface [18]. The Catamount system does not support threads and can thus not run the threaded implementation. An unreleased development version of Open MPI [16] was used to perform these measurements, as Open MPI 1.2.1 does not provide Portals communications support. However, using the NIC-supported overlap with LibNBC results in a better overall system usage and an up to 14.2% higher parallel efficiency of the FFT on 128 processes.

Another benchmark on the Coyote system (cf. 4), shown on Fig. 4, shows results for runs of the 1024^3 FFT box transformation on 128 processes with

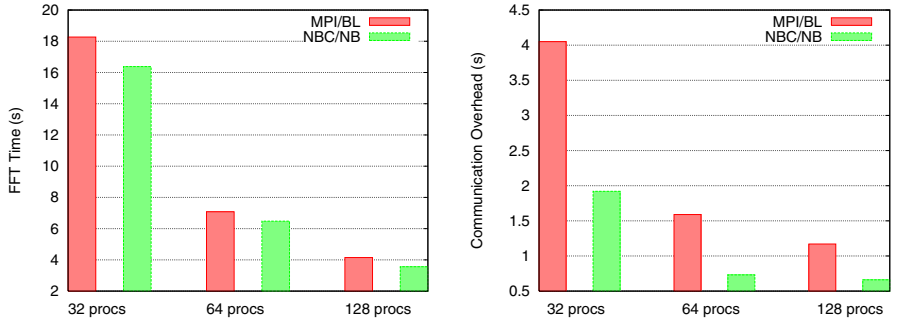


Fig. 3. Left: Blocking and non-blocking FFT times for different process counts on the XT4 system. Right: Communication overhead.

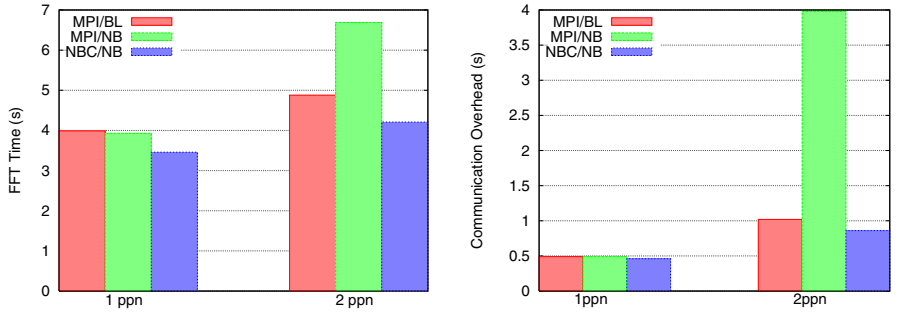


Fig. 4. Left: Blocking and non-blocking FFT times for different node configurations of 128 processes on the Coyote system (using 128 or 64 nodes respectively). Right: Communication overhead.

either 1 process per node (1ppn) or two processes per node (2ppn). This effectively compares the efficiency of the MPI approach (perform the non-blocking collectives in a separate thread) with the LibNBC approach (use non-blocking point-to-point communication). We clearly see the the LibNBC approach is superior on this system. As soon as all available CPUs are used for computation, the threaded approach even slows the execution down (cf. Section 4). Our conclusion is that with the currently limited number of CPU cores, it does not pay off to invest half of the cores to process asynchronous collectives with the MPI approach; they should rather be used to perform useful computation. Thus, we suggest the usage of non-blocking point-to-point as in LibNBC.

6 Conclusions

As modern computing and communication hardware is becoming more powerful, it is providing more opportunities for delegation of communication operations

and hiding of communication costs. MPI has long supported asynchronous point-to-point operations to take advantage of these capabilities. It is clearly time for the standard to support non-blocking functionality for collective operations.

The interface we propose is a straightforward extension of the current MPI collective operations and we have implemented a prototype of these extensions in a library using MPI point-to-point operations. We note however, that implementing non-blocking collective operations as a separate library in this way requires implementing (potentially quite similar) collective algorithms in two different places (the blocking and non-blocking cases). Having those operations standardized in MPI would enable a single shared infrastructure inside the MPI library. In addition, communicator duplication is necessary in both implementations and can not be done in a non-blocking way without user interaction.

Our results with a microbenchmark and an application clearly show the performance advantages of non-blocking collectives. In the case of an all-to-all communication, we are able to overlap up to 99% of the communication with user computation on our systems. The application of a pipelined computation/communication scheme to a 3d-FFT shows application performance gains for a 128 process job of up to 14.2% on a Cray XT4 and 13.7% on an InfiniBand-based cluster system. In particular, we show that using the MPI-2 threaded model for a real-world problem to perform non-blocking collective operations is clearly suboptimal to an implementation based on non-blocking point-to-point operations.

Acknowledgements

Pro Siobhan. The authors want to thank Laura Hopkins from Indiana University for editorial comments and helpful discussions. This work was supported by a grant from the Lilly Endowment and National Science Foundation grant EIA-0202048. This research was funded in part by a gift from the Silicon Valley Community Foundation, on behalf of the Cisco Collaborative Research Initiative of Cisco Systems. This research was also sponsored in part by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

1. Brightwell, R., Riesen, R., Underwood, K.D.: Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. Int. J. High Perform. Comput. Appl. 19(2), 103–117 (2005)
2. Danalis, A., Kim, K.Y., Pollock, L., Swany, M.: Transformations to parallel codes for communication-computation overlap. In: SC 2005, p. 58. IEEE Computer Society, Washington, DC, USA (2005)
3. Gropp, W.D., Thakur, R.: Issues in developing a thread-safe mpi implementation. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)

4. Kale, L.V., Kumar, S., Vardarajan, K.: A Framework for Collective Personalized Communication. In: Proceedings of IPDPS'03, Nice, France (April 2003)
5. Dubey, A., Tessera, D.: Redistribution strategies for portable parallel FFT: a case study. Concurrency and Computation: Practice and Experience 13(3), 209–220 (2001)
6. Kanevsky, A., Skjellum, A., Rounbehler, A.: MPI/RT - an emerging standard for high-performance real-time systems. HICSS (3), 157–166 (1998)
7. Hoefler, T., Gottschling, P., Rehm, W., Lumsdaine, A.: Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 374–382. Springer, Heidelberg (2006)
8. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and performance analysis of non-blocking collective operations for mpi. In: Submitted to Supercomputing'07 (2007)
9. Adelmann, A., Bonelli, A., Ueberhuber, W.P.P.C.W.: Communication efficiency of parallel 3d ffts. In: Daydé, M., Dongarra, J.J., Hernández, V., Palma, J.M.L.M. (eds.) VECPAR 2004. LNCS, vol. 3402, pp. 901–907. Springer, Heidelberg (2005)
10. Calvin, C., Desprez, F.: Minimizing communication overhead using pipelining for multidimensional fft on distributed memory machines (1993)
11. Goedecker, S., Boulet, M., Deutsch, T.: An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes. Computer Physics Communications 154, 105–110 (2003)
12. Hoefler, T., Squyres, J., Bosilca, G., Fagg, G., Lumsdaine, A., Rehm, W.: Non-Blocking Collective Operations for MPI-2. Technical report, Open Systems Lab, Indiana University (08, 2006)
13. LibNBC (2006), http://www.unixer.de/NBC
14. Hoefler, T., Lumsdaine, A.: Design, Implementation, and Usage of LibNBC. Technical report, Open Systems Lab, Indiana University (08 2006)
15. Vadhiyar, S.S., Fagg, G.E., Dongarra, J.: Automatically tuned collective communications. In: Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), p. 3. IEEE Computer Society, Washington, DC, USA (2000)
16. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (September 2004)
17. Alverson, R.: Red storm. Invited Talk, Hot Chips 15 (2003)
18. Brightwell, R., Hudson, T., Maccabe, A.B., Riesen, R.: The portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories (1999)

Optimization of Collective Communications in HeteroMPI

Alexey Lastovetsky, Maureen O'Flynn, and Vladimir Rychkov

School of Computer Science and Informatics, University College Dublin,
Belfield, Dublin 4, Ireland
{Alexey.Lastovetsky,Maureen.OFlynn}@ucd.ie
http://hcl.ucd.ie

Abstract. HeteroMPI is an extension of MPI designed for high performance computing on heterogeneous networks of computers. The recent new feature of HeteroMPI is the optimized version of collective communications. The optimization is based on a novel performance communication model of switch-based computational clusters. In particular, the model reflects significant non-deterministic and non-linear escalations of the execution time of many-to-one collective operations for medium-sized messages. The paper outlines this communication model and describes how HeteroMPI uses this model to optimize one-to-many (scatter-like) and many-to-one (gather-like) communications. We also demonstrate that HeteroMPI collective communications outperform their native counterparts for various MPI implementations and cluster platforms.

Keywords: MPI, HeteroMPI, heterogeneous cluster, switched network, message passing, collective communications, scatter, gather.

1 Introduction

MPI [1] is the most widely used programming tool for parallel computing on distributed-memory computer systems. It can be used on both homogeneous and heterogeneous clusters, but it does not provide specific support for development of high performance parallel applications for heterogeneous networks of computers (HNOC).

HeteroMPI [2] is an extension of MPI designed for high performance computing on HNOCs. It supports optimal distribution of computations among the processors of a HNOC by taking into account heterogeneity of processors, network topology and computational costs of algorithm. The main idea of HeteroMPI is to automate the creation of a group of processes that will execute the heterogeneous algorithm faster than any other group. It is achieved by specifying the performance model of the parallel algorithm and by optimal mapping of the algorithm onto the HNOC, which is seen by the HeteroMPI programming system as a multilevel hierarchy of interconnected sets of heterogeneous multiprocessors. HeteroMPI is implemented on top of MPI, therefore it can work on top of any MPI implementation. HeteroMPI introduces a small number of

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 135–143, 2007.

additional functions for group management and data partitioning. All standard MPI operations are inherited, so that the existing MPI programs can be easily transformed into HeteroMPI.

HeteroMPI inherits all MPI communication operations and solely relies on their native implementation. At the same time, our recent research on the performance of MPI collective communications on heterogeneous clusters based on switched networks [3] shows that MPI implementations of scattering and gathering are often very far from optimal. In particular, we observed very significant escalations of the execution time of many-to-one MPI communications for medium-sized messages. The escalations are non-deterministic but form regular levels. We also observed a leap in the execution time of one-to-many communications for large messages. Based on the observations, we suggested a communication performance model of one-to-many and many-to-one MPI operations reflecting these phenomena [3], which is applicable to both heterogeneous and homogeneous clusters. The paper presents a new feature of HeteroMPI, which is the optimized version of collective communication operations. The design of these optimized operations is based on the new communication model and can be summarized as follows:

- Upon installation, HeteroMPI computes the parameters of the model.
- Each optimized collective operation is implemented by a sequence of calls to native MPI operations. The code uses parameters of the communication model.

This high-level model-based approach to optimization of MPI communications is easily and uniformly applied to any combination of MPI implementation and cluster platform. It does not need to retreat to the lower layers of the communication stack and tweak them in order to improve the performance of MPI-based communication operations. This is particularly important for heterogeneous platforms where the users typically have neither authority nor knowledge for making changes in hardware or basic software settings.

The paper is structured as follows. Section 2 outlines the related work. Section 3 briefly introduces the communication model. Section 4 describes the implementation of the optimized collective operations in HeteroMPI. Section 5 presents experimental results, demonstrating that HeteroMPI collective communications outperform their native counterparts for different MPI implementations and clusters platforms.

2 Related Work

Vadhiyar et al. [4] developed automatically tuned collective communication algorithms. They measured the performance of different algorithms of collective communications for different message sizes and numbers of processes and then used the best algorithm. Thakur et al. [5] used a simple linear cost model of a point-to-point single communication in selection of algorithms for a particular collective communication operation. Pjesivac-Grbovic et al. [6] applied different point-to-point models to the algorithms of collective operations, compared

the predictions with measurements and implemented the optimized versions of collective operations based on the decision functions that switch between different algorithms, topologies, and message segment sizes. Kielmann et al. [7] optimized MPI collective communication for clustered wide-area environments by minimizing communication over slow wide-area links. There were some works on improving particular MPI operations [8,9].

All works on the optimization of collective operations are based on deterministic linear communication models. Implementation of the optimized versions of collective operations in HeteroMPI uses the performance model that takes into account non-deterministic escalations of the execution time of many-to-one MPI communications for medium-sized messages and the leap in the execution time of one-to-many communications for large messages.

3 The Performance Model of MPI Communications

This section briefly introduces the new performance model of MPI communications [3], which reflects the phenomena observed for collective communications on clusters based on switched networks. The basis of the model is a LogP-like [11] model of point-to-point communications for heterogeneous clusters [10]. The parameters of the point-to-point model represent the heterogeneity of processors and are also used in construction of the models of collective communications. Apart from the point-to-point parameters, the models of collective communications use parameters reflecting the observed non-deterministic and non-linear behavior of MPI collective operations.

Like any other ***point-to-point communication model***, except for PLogP our model is linear, representing the communication time by a linear function of the message size. The execution time of sending a message of M bytes from processor i to processor j on heterogeneous cluster is estimated by $C_i + Mt_i + C_j + Mt_j + \frac{M}{\beta_{ij}}$, where C_i, C_j are the fixed processing delays; t_i, t_j are the delays of processing of a byte; β_{ij} is the transmission rate. Different parameters for nodal delays reflect heterogeneity of the processors. For networks with a single switch, it is realistic to assume $\beta_{ij} = \beta_{ji}$.

There are two components in the models of one-to-many and many-to-one communications. The first one is built upon the model of point-to-point communications by representing each collective communication, MPI_Scatter or MPI_Gather, by a straightforward combination of point-to-point communications

```
if (rank==root) {
  memcpy(recvbuf, sendbuf, recvcount);
  for (i=1; i<n; i++) {
    if (scatter)
      MPI_Isend(sendbuf+sendcount*i, sendcount, i);
    if (gather)
      MPI_Irecv(recvbuf+recvcount*i, recvcount, i);
  }
```

```
  MPI_Waitall(n-1);
}
else {
  if (scatter)
    MPI_Recv(recvbuf, recvcount, root);
  if (gather)
    MPI_Send(sendbuf, sendcount, root);
}
```

This approach obviously results in a linear predictive model, which is quite accurate for relatively small message sizes but does not reflect the observed phenomena of non-deterministic and non-linear escalations of the execution time of many-to-one communications for medium-sized messages and the significant leap in the execution time of one-to-many communications for large messages. The second component in the collective communication models addresses the issues.

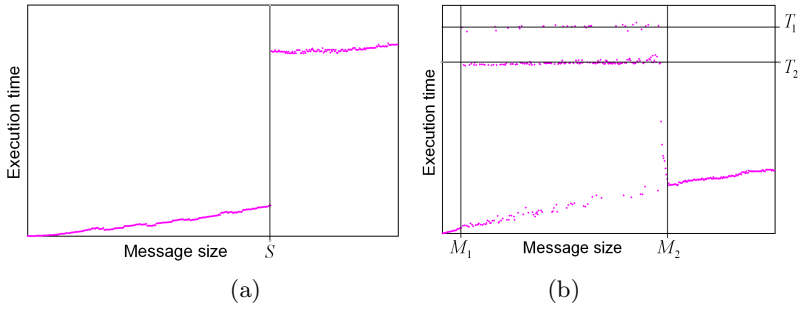


Fig. 1. The execution time of collective communications against the message size: (a) one-to-many and (b) many-to-one

Fig. 1 shows the typical behavior of one-to-many and many-to-one communications on a switch-based cluster, given the operations are implemented via the described straightforward combination of point-to-point communications. One can see a distinctive leap in the execution time for the one-to-many operation as well as non-deterministic and non-linear escalations of the execution time of the many-to-one operation for medium-sized messages. These phenomena were observed on the MPI implementations over TCP communication layer but not over Myrinet-MX. So, they may be caused by some TCP features. At the same time, we have not had a chance to experiment with a reasonably large Myrinet-based cluster and, therefore, cannot guarantee that MPI over Myrinet-MX does not have such irregularities.

The ***one-to-many model*** [10,3] reflects the leap in the execution time and categorizes the small and large messages. Parameter S is a message size threshold, separating small and large messages. It is different for different combinations of clusters and MPI implementations. The estimated time of scattering

messages of size M from node 0 to nodes $1, 2, ..., n$ is given by $C_0 + t_0 \times n \times M + \max_{1 \le i \le n} \left\{ C_i + t_i M + \frac{M}{\beta_{0i}} \right\}$, if $M < S$ or $C_0 + t_0 \times n \times M + \sum_{i=1}^{n} (C_i + t_i M + \frac{M}{\beta_{0i}})$, if $M \ge S$, where C_0, t_0, C_i, t_i are the fixed and variable processing delays on the source node and destinations. The one-to-many model displays parallel communication for small messages and a serialized communication for large messages.

The ***many-to-one model*** [3] differentiates small, medium and large messages by introducing parameters M_1 and M_2. For small messages, $M < M_1$, the execution time has a linear response to the increase of message size. Thus, the execution time for the many-to-one communication involving n processors ($n \le N$, where N is the cluster size) is estimated by $n(C_0 + t_o M) + \max_{1 < i \le n} \left\{ C_i + t_i M + \frac{M}{\beta_{0i}} \right\} + \kappa_1 M$, where κ_1 is a fitting parameter for correction of the slope. For large messages, $M > M_2$, the execution time resumes a linear predictability for increasing message size. Hence, this part of the model has the same design but a different slope of linearity and greater value due to overheads: $n(C_0 + t_0 M) + \sum_{i=1}^{n} (C_i + t_i M + \frac{M}{\beta_{0i}}) + \kappa_2 M$. The additional parameter κ_2 is a fitting constant for correction of the slope. For medium messages, $M_1 \le M \le M_2$, we observed a small number of discrete levels of escalation, remaining constant as the message size increases. The model describes the probability of escalation to each of the levels as a function of message size and the number of processors involved in the operation. If no escalation occurs, the linear model used for small massages will accurately predict the execution time.

The presented model accurately describes the performance of many-to-one and one-to-many operations for all combinations of MPI implementations and cluster platforms, which we used in our experiments, if the collective operations were implemented via point-to-point MPI operations (as described by the pseudo-code above). For LAM [12] and Open MPI [13], MPI_Scatter and MPI_Gather display exactly the same performance pattern as their straightforwardly implemented counterparts. Therefore, such MPI implementations need no further extension of the communication model in order to describe native collective communication operations. At the same time, for some MPI implementations (mainly, some versions of MPICH [14]), the native collective communications perform differently (better or worse) than their straightforward counterparts. To deal with such MPI implementations, HeteroMPI uses an extended communication model, additionally including a separate model for each native collective operation. Due to space limitations, we do not include in the paper considerations related to this extended model.

In implementation of the optimized scatter and gather collective operations, we use the message size thresholds S, M_1 and M_2 to fragment the messages and to avoid the message sizes for which the irregularities are observed. These parameters are found experimentally for a parallel platform. The building of the analytical part of the communication model is out of the scope of this paper. We do not describe the communication experiments and the measurement techniques required to find the rest of parameters.

4 Optimization of Collective Operations in HeteroMPI

This section describes the implementation of two newly introduced HeteroMPI operations, HMPI_Scatter and HMPI_Gather, which are optimized versions of native MPI_Scatter and MPI_Gather respectively. The implementation uses the communication performance model presented in Section 3 in order to avoid the MPI_Gather time escalations and the MPI_Scatter leap in the execution time. More precisely, only the message size thresholds S, M_1 and M_2 are used in the implementation. These parameters are computed by the HeteroMPI programming system upon its installation on the parallel platform. In the implementation, neither point-to-point nor low-level communications are used, but only the native MPI counterparts.

The implementation of HMPI_Gather re-invokes the native MPI_Gather for small and large messages. The gathering of medium-sized messages, $M_1 \leq M \leq M_2$, is implemented by an equivalent sequence of m MPI_Gather operations with messages of the size that fits into the range of small messages: $\frac{M}{m} < M_1$ and $\frac{M}{m-1} \geq M_1$. Small-sized gatherings are synchronized by barriers, which removes communication congestions on the receiving node. The barriers are marked bold in the pseudo code:

```
if (M1<=M<=M2) {
  find m such that M/m<M1 and M/(m-1)>=M1;
  for (i=0; i<m; i++) {
    MPI_Barrier(comm);
    MPI_Gather(sendbuf + i*M/m, M/m);
  }
}
else MPI_Gather(sendbuf, M);
```

Note. If MPI_Barrier is removed from the code, the resulting implementation will behave exactly as the native MPI_Gather. It means that this synchronization is essential for preventing communication congestions on the receiving side.

The implementation of HMPI_Scatter uses parameter S of one-to-many communication model. For small messages, $M < S$, the native MPI_Scatter is re-invoked. The scattering of large messages is implemented by an equivalent sequence of MPI_Scatter operations with messages of the size less then S: $\frac{M}{m} < S$ and $\frac{M}{m-1} \geq S$. The pseudo code of the optimized scatter is as follows:

```
if (M>=S) {
  find m such that M/m<S and M/(m-1)>=S;
  for (i=0; i<m; i++)
    MPI_Scatter(recvbuf + i*M/m, M/m);
}
else MPI_Scatter(recvbuf, M);
```

As the presented approach does not use the communication parameters reflecting the heterogeneity of the processors, it can be applied to both homogeneous and heterogeneous switch-based clusters.

5 Experiments

To compare the performance of the optimized HeteroMPI collective operations with their native MPI counterparts, we experimented with various MPI implementations and different clusters. Here we present the results for the following two platforms:

- **LAM-Ethernet:** 11 x Xeon 2.8/3.4/3.6, 2 x P4 3.2/3.4, 1 x Celeron 2.9, 2 x AMD Opteron 1.8, Gigabit Ethernet, LAM 7.1.3,
- **OpenMPI-Myrinet:** 64 x Intel EM64T, Myrinet, Open MPI 1.2.2 over TCP.

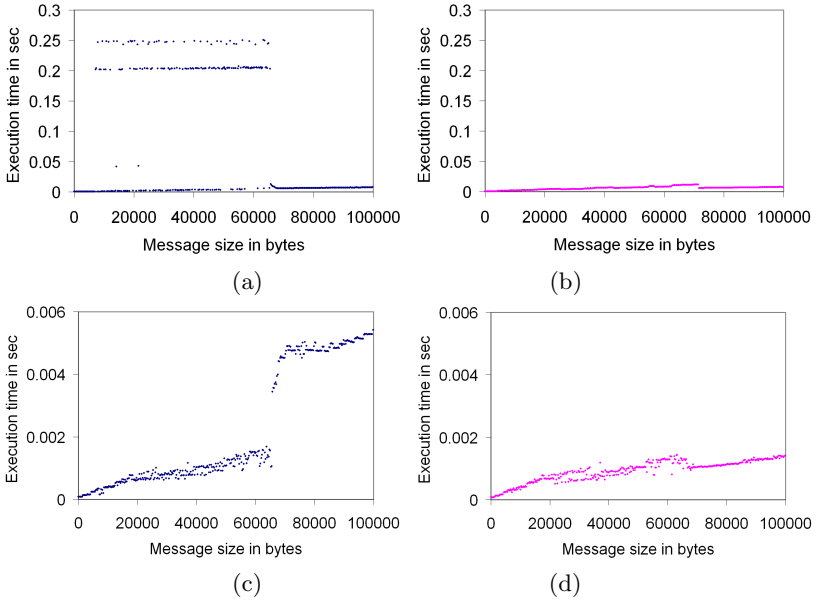


Fig. 2. Performance of (a) MPI_Gather, (b) HMPI_Gather, (c) MPI_Scatter, (d) HMPI_Scatter on 16-nodes heterogeneous cluster LAM-Ethernet

Fig. 2 shows the results for the heterogeneous LAM-Ethernet cluster, with all nodes in use. The message size thresholds for this platform are $M_1 = 5KB$, $M_2 = 64KB$, $S = 64KB$. Similar results are obtained on the 64-node homogeneous OpenMPI-Myrinet cluster (Fig. 3). For this platform, $M_1 = 5KB$, $M_2 = 64KB$, $S = 64KB$. The results show that the optimized HeteroMPI versions outperform their native MPI counterparts, avoiding the escalations and

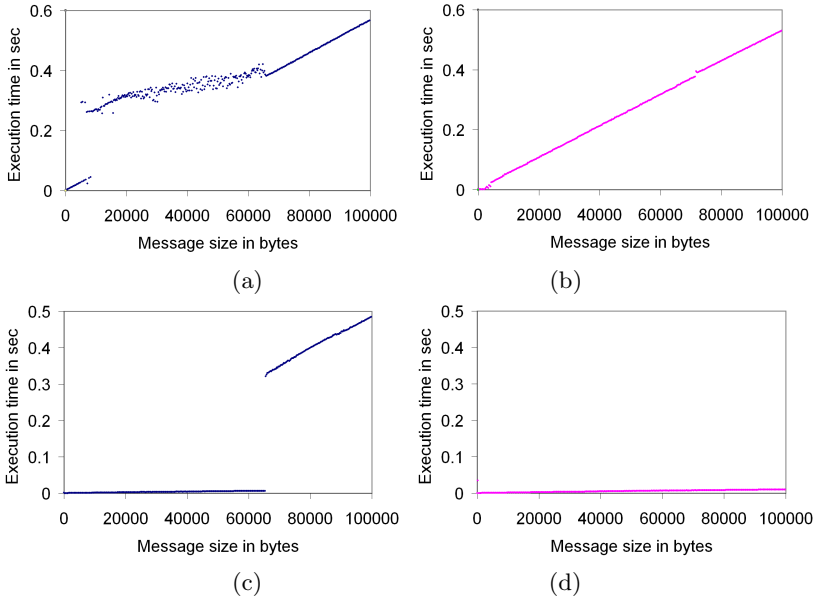


Fig. 3. Performance of (a) MPI_Gather, (b) HMPI_Gather, (c) MPI_Scatter, (d) HMPI_Scatter on 64-nodes OpenMPI-Myrinet cluster

restoring the linear dependency of the communication execution time on message size. On all platforms we observed $S = M_2$.

The communication execution time was measured on the root node. The barrier was used to ensure that all processes have finished the scatter-like operations. The communication experiments for each message size in a series were carried out only once. The repeated measurements gave similar results. To avoid the pipeline effect in a series of the experiments for different message sizes, the barriers were included between collective operations.

6 Conclusion

The paper introduced a new feature of HeteroMPI, which is the optimized versions of MPI collective communications for switched-based computational clusters. The optimized collective operations were implemented on top of the corresponding MPI functions and based on the communication performance model. We also presented experimental results demonstrating that the optimized functions outperformed the native ones.

The proposed approach to optimization of MPI communications is based on the use of a high-level communication performance model. Therefore, it can be easily and uniformly applied to any combination of MPI implementation and cluster platform. It needs no retreat to the lower layers of the communication stack for tweaking them in order to improve the performance of MPI-based

communication operations. This is particularly advantageous for heterogeneous platforms where the users typically have neither the authority nor the knowledge for changing hardware and basic software settings.

Acknowledgments. The work was supported by the Science Foundation Ireland (SFI). We are grateful to the Innovative Computing Laboratory, University of Tennessee, for providing with computing clusters.

References

1. Dongarra, J., Huss-Lederman, S., Otto, S., Snir, M., Walker, D.: MPI: The Complete Reference. The MIT Press, Cambridge (1996)
2. Lastovetsky, A., Reddy, R.: HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. J. of Parallel and Distr. Comp. 66, 197–220 (2006)
3. Lastovetsky, A., O'Flynn, M.: A Performance Model of Many-to-One Collective Communications for Parallel Computing. In: Proc. of IPDPS 2007, Long Beach, CA (2007)
4. Vadhiyar, S.S., Fagg, G.E., Dongarra, J.: Automatically tuned collective communications. In: Proc. of Supercomputing 99, Portland, OR (1999)
5. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of Collective Communication Operations in MPICH. Int. J. of High Perf. Comp. App. 19, 49–66 (2005)
6. Pjesivac-Grbovic, J., Angskun, T., Bosilca, G., Fagg, G.E., Gabriel, E., Dongarra, J.J.: Performance Analysis of MPI Collective Operations. In: Proc. of IPDPS 2005, Denver, CO (2005)
7. Kielmann, T., Hofman, R.F.H., Bal, H., Plaat, A., Bhoedjang, R.A.F.: MagPIe: MPI's collective communication operations for clustered wide area systems. In: Proc. of PPoPP 1999, pp. 131–140. ACM Press, New York (1999)
8. Iannello, G.: Efficient algorithms for the reduce-scatter operation in LogGP. IEEE Transactions on Parallel and Distr. Systems 8(9), 970–982 (1997)
9. Benson, G.D., Chu, C-W., Huang, Q., Caglar, S.G.: A comparison of MPICH allgather algorithms on switched networks. In: Dongarra, J.J., Laforenza, D., Orlando, S. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2840, pp. 335–343. Springer, Heidelberg (2003)
10. Lastovetsky, A., Mkwawa, I., O'Flynn, M.: An Accurate Communication Model of a Heterogeneous Cluster Based on a Switch-Enabled Ethernet Network. In: Proc. of ICPADS 2006, Minneapolis, MN, pp. 15–20 (2006)
11. Culler, D., Karp, R., Patterson, R., Sahay, A., Schauser, K.E., Santos, R.S.E., von Eicken, T.: LogP: Towards a realistic model of parallel computation. In: Proc. of the 4th ACM SIGPLAN, ACM Press, New York (1993)
12. LAM/MPI User's Guide, http://www.lam-mpi.org/
13. Open MPI Publications http://www.open-mpi.org/
14. MPICH/MPICH-2 User's Guide, http://www-unix.mcs.anl.gov/mpi/

Low Cost Self-healing in MPI Applications

Jacques A. da Silva and Vinod E. F. Rebello

Instituto de Computação, Universidade Federal Fluminense, Brazil
{jacques, vinod}@ic.uff.br

Abstract. Writing applications capable of executing efficiently in Grids is extremely difficult and tedious for inexperienced users. The distributed resources are typically heterogeneous, non-dedicated, and are offered without any performance or availability guarantees. Systems capable of adapting the execution of an application to the dynamic characteristics of the Grid are essential. This work describes the strategy used to bestow the self-healing property on autonomic EasyGrid MPI applications to withstand process and resource failures. This paper highlights both the difficulties and the low cost solution adopted to offer fault tolerance in applications based on the standard Grid installation of LAM/MPI.

Keywords: Fault tolerance, Autonomic MPI, Grid middleware.

1 Introduction

Computational Grids aim to aggregate significant numbers of geographically distributed resources to provide sufficient (and low cost) computational power to an ever growing variety of applications. Given the scale and shared nature of Grids, they are especially vulnerable to execution and communication failures. Many e-science applications require to run for days or weeks at a time which is becoming significantly longer than the mean time between failures of many grid environments. The inability to tolerate faults not only degrades performance since applications will have to be restarted, but wastes valuable computing power.

Given the characteristics of parallel applications and the dynamism and heterogeneity of Grids, the need for systems capable of adapting the execution of applications to the environment is clear. Autonomic computing [1] focuses on the design of systems and applications that regulate themselves. Initially four general properties, together with four enabling attributes, were defined as requirements for these self-managing systems: self-configuring, self-healing, self-optimizing, self-protecting, self-awareness, environment-awareness, self-monitoring and self-adjusting [2]. The development of mechanisms to enable these systems and applications relies on autonomic elements i.e., embedded self-contained modules responsible for implementing their functions and behaving in accordance with the context and policies defined or deployed at run time. Although self-healing (the ability to detect and recover from faults and continue executing correctly) appears to be identical to fault tolerance, its efficient implementation is dependent on an effective integration with other self-* mechanisms, in particular self-monitoring, self-adjusting and self-optimization.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 144–152, 2007.

The EasyGrid Application Management System (AMS) transforms cluster-based MPI applications (designed for homogeneous, stable environments) into autonomic ones capable executing robustly and efficiently in Grids [3]. The EasyGrid AMS is an application-specific application-level middleware that is embedded into the user's MPI program at compile time, thus creating a self-contained autonomic MPI application. This paper describes and evaluates the integrated self-healing mechanism adopted by the AMS to permit EasyGrid MPI applications to complete their execution in Grids even in the face of failures. The EasyGrid AMS adopts the "one MPI process per application task" grid execution model [4] and therefore is based on MPI implementations which support dynamic process creation such as LAM/MPI. For reasons of deployability, no modifications are made standard distribution or installation of LAM/MPI and no further software, other than the Globus ToolKit 2.x, is required for execution of autonomic EasyGrid MPI applications in a Grid. We assume that while MPI provides a reliable message delivery service, processes and/or resources can fail at any time. The objective is to efficiently integrate low cost fault tolerance and scheduling strategies into MPI programs, appropriate for the proposed grid execution model, in a way that is transparent to the programmer.

2 Concepts and Related Work

Normally, fault tolerance mechanisms have 4 stages: fault detection is the process of recognizing that an error has occurred; fault location then involves identifying the location of the component of the system that caused the error; fault containment aims to prevent a possible propagation of the fault to the rest of the system by isolating the faulty component; and fault recovery is the process of restoring the operational status of the system to a previous fault-free consistent state. To avoid having to re-execute the entire application, fault recovery is generally based on checkpointing, message logging or a combination of both [5]. Checkpointing techniques periodically save the execution state of the processes. If a failure is detected, all processes are rolled back to a previous checkpoint and restarted. Checkpointing techniques can be both complex (since the parallel application needs to be rolled back to a consistent state) and expensive (since fault free storage is required and which, if not distributed, can be subject to congestion when writing and retrieving checkpoints). Message-logging, on the other hand, aims to restart the failed process only. Although the other processes are not rolled back, all messages sent previously to the process that failed must be resent. The drawback is the cost associated with keeping a copy of all messages sent and managing communication to and from the replacement process. Checkpointing may be used in conjunction to reduce the size of the message log.

MPI was initially designed for homogeneous, fault-free, static environments such as dedicated computing clusters. With computing systems become larger, these characteristics are changing and researchers have begun, in particular, to explore diverse approaches to offer fault-tolerance in MPI. Of course, the strategies proposed are confined by the execution model adopted by the application.

MPI applications are typically designed to execute long running processes, one per processor [6]. In dynamic heterogeneous environments like Grids, this *one process per processor* (1PProc) execution model becomes inappropriate since the granularity of each process tend to require adjustments. The fact that resources maybe heterogeneous, shared with local jobs, and fault-prone not only makes this model inefficient but also makes managing the execution of applications extremely complex. A *one process per task* (1PTask) execution model has been proposed where programs consist of a larger number (determined by the parallelism of the application and not the number of resources) of shorter-lived processes [4]. For this model, we propose to adopt a fault tolerance strategy based solely on message logging. The benefits gained from avoiding the need to implement sophisticated checkpointing schemes and maintain long message logs, can outweigh the cost of managing an increased number of processes.

The MPICH-V [7] is a fault-tolerant version of MPICH that implements uncoordinated checkpointing and logging of messages to allow aborted or failed processes to be substituted. All comunications first pass through Channel Memories (executing on remote persistent resources) for logging and thus incurs an overhead for message delivery. MPI-FT [8] uses one of two logging strategies for recovery. Communications can take place via a central observer process. Should a process fail, messages are redirected to a replacement. Alternatively, each process is responsible for keeping a copy of all messages it sent. In the case of failure, the observer indicates to whom messages should be resent. FT-MPI [9] does not checkpoint or log messages. Instead it extends the functionality of the communicators to provide information which will permit an application to take the appropriate corrective action. The drawback is the lack of transparency for the programmer.

A library `EGAMSmpi.h` has been developed to substitute `mpi.h`, which activates EasyGrid AMS management mechanisms through the code wrapping of MPI functions called by the application. This functionality is included into the user's program at compile time, without the need of changes to the source code. Furthermore, the EasyGrid AMS offers more than just self-healing. Fault tolerance has been integrated with mechanisms which support the other autonomic self-* properties to provide an effective low instrusion self-management system.

3 EasyGrid Application Management System

Parallel MPI applications are transformed into adaptive, fault tolerance, and self-scheduling programs capable of harnessing available Grid resources efficiently by coupling each of them with an application-specific middleware in the form of an application management system (AMS). The AMS controls the execution of the MPI application processes through a distributed hierarchy of management processes. This hierarchical structure is composed by three management levels. At the top, a single Global Manager (GM) is responsible for supervising the sites in the Grid where the application's processes are running (or will be able to execute). The next level is composed of Site Manager (SM) processes that manage

the execution at individual sites. Finally, at the lowest level, Host Managers (HM), one for each resource, take on the responsibility for scheduling, creating and executing the application processes allocated to their respective host.

Each management process is designed following a layered subsumption architecture. The functionality of each layer depends on the management process' level in the hierarchy. This hierarchical structure allows the application to adapt independently to environmental changes, since each management process can adopt differing dynamic policies. The process management layer is responsible for the dynamic creation of MPI processes, both of the application and management ones, and for the redirection of messages between processes. The application's self-monitoring layer collects system data and provides status information for re-scheduling the application processes and recovering from process failures.

Although, the EasyGrid AMS adopts the 1PTask execution model, not all application processes are created at start up. All of the MPI processes (except GM), including those of the application, are created dynamically according to the hierarchical management structure - each child with a unique communicator. Since the original application processes can no longer communicate directly with each other, the HMs, SMs and GM must take on the job of routing messages between application processes in the same host, same site, and between sites, respectively. Messages must also be stored for processes that have yet to be created. For fault tolerance, we extend the life time of each process' input message log until that process successfully completes. The SM's are also responsible for the eventual communication between the HMs and the GM, re-directing not only application messages, but also AMS messages (e.g. error and monitoring). Note that the management processes need to route messages dynamically since destination processes may have been reallocated by the self-optimizing dynamic scheduler in response to self-healing (process failures) or changes in the compute power available from resources.

4 Fault Tolerance in the EasyGrid AMS

The AMS's management of processes permits the recovery from *process* and *processor* failures without need to interrupt the execution of the application. The AMS can tolerate failures in all processes except the GM. Since the AMS offers application-level fault tolerance (FT) over standard LAM/MPI, the GM must either have FT support provided by the system which started the application or be created on a persistent resource (the existence of which is a common requisite of existing FT strategies). The AMS uses distinct inter-communicators between each pair of processes (for both management and application processes) so that faults can easily located and isolated. This leaves fault detection and fault recovery to be addressed.

The AMS fault tolerance subsystem detects faults through communication errors in the monitoring subsystem, i.e. when messages are being sent to or received from the application processes by the HM and between the management

processes. Both the error type and the identification of the process that cause the fault are obtained since error handlers are associated with communicators.

However, detecting MPI process failures is tricky, especially since non-blocking MPI calls are local operations and do not always identify that a remote process has died even with error handlers set to `MPI_ERRORS_RETURN`. Furthermore, since the order, size and arrival times of messages are unknown, the management processes use such calls to minimise intrusion. Another issue that has been observed is that the sending of two consecutive messages using the eager protocol (i.e. for message sizes between 52105 and 65536 bytes) to a failed process can cause the sender to die. For this reason, the fault tolerance mechanisms developed were required to test the state of the remote process prior to the sending of a message.

We use the term *hardware failure* to refer to a non-recoverable fault in a resource (including shutdown) or the failure of the lamd daemon (the recreation of a new daemon is not possible with the Globus module of LAM/MPI). A *software failure* occurs when a management or application process fails. Since the resource continues to be available, the application should recover to take advantage of it. We proposed a distributed but unified approach to simultaneously detect both hardware and software faults in MPI applications based on LAM/MPI.

4.1 Fault Tolerance for Application Processes

The responsibility to detect and recover from failures that occur in the application processes lies with the HM that created the process. The technique employed for detecting such failures was presented in [3]. The AMS implements message logs to hold the input messages of each application process until that process executes successfully. The allocated HM logs messages chronologically. This scheme assume that processes are deterministic, i.e. the same inputs always produce the same output. If the elapsed time since the last monitoring message received from a local process v exceeds a predetermined threshold, a signal 0 is sent to test the communicator using the `MPIL_Signal` call. Should the function return an error, a fault is detected and the communicator between the manager and process is freed to prevent a possible propagation of the fault. The HM scheduling subsystem is instructed to create a new process $v\prime$ to substitute v. All messages for $v\prime$ will be recovered from the v's message log held locally by the HM. Note that the resending of messages by $v\prime$ that were delivered sucessfully prior to the failure are discarded by the HM that created $v\prime$.

4.2 Fault Tolerance for Host Managers

The SM is responsible for guaranteeing fault tolerance with respect to failures on other resources at its site. The SM's fault tolerance mechanism must interact with its scheduler to redistribute application processes among the other HM's in the case of hardware failures. Adopting the previous technique is not an option because the function `MPIL_Signal` becomes blocked when a remote daemon has died. This would cause the SM to stall when a HM becomes unreachable.

The solution adopted uses threads to execute a system call with the command *mpitask* to test the HMs. The command mpitask can be used to monitor MPI processes in the LAM environment. Although the command mpitask also blocks if a remote daemon has died, the thread can be cancelled and MPI functions will continue functioning normally. Peculiarly however, if the command mpitask is killed 8 times, the lamd daemon on the resource executing the SM loses its connectivity with the other daemons. To prevent this, the command *lamshrink* is used to remove the faulty node from the LAM universe.

As mentioned earlier, a process may die if it sends two messages to a process which has already died. Therefore, the AMS verifies the status of each destination process prior to communicating. However to hide this overhead, the status of a process is only requested after each communication and verified just before the next communication to that process. Even though a failure may occur after the test but before the communication, this will be detected before the second message is sent. The mechanism to detect faults consists of creating a thread to test if the daemon and the HM on the remote node are alive. A variable shared between this thread and the main thread (the SM) receives the result of the mpitask call. If this result is not available when the SM needs to send another message, the SM will continue testing the variable after a series of exponentially increasing timeouts. If the thread is not blocked, i.e. mpitask completed, then the remote daemon is alive. The shared variable will identify if a fault has occurred in the HM. After a final timeout, if thread is still blocked, then the SM assumes that the daemon on the remote host has died and the SM executes *lamshrink*. In the case where the HM dies but daemon remains alive, the SM will recreate the HM and restore its previous state. The SM keeps a message log for all application processes currently scheduled in its site. When created, the HM receives a list of application processes and the messages they received previously. If the daemon has died, the resource is no longer considered usable and the application processes that were allocated to that resource are redistributed. This operation is carried out by the dynamic scheduler and does not involve process migration.

4.3 Fault Tolerance in Site Managers

GM is the responsible for guaranteeing fault tolerance in the SMs and scheduling tasks between the sites [3]. The fault detection and recovery scheme is identical to that used by the SMs. The GM is the only process that possesses information on all of the application processes. However, due to the use of distinct inter-communicators, when a SM dies, the HMs created by this SM become orphans. When a HM detects that its parent SM has died, it terminates the execution of its application processes and exits, thus freeing the resource. A recreated SM initiates its execution by recreating the HMs at its site. Here we assume that if a SM's resource suffers a hardware failure, the site is inaccessible and all application processes at that site must be redistributed amongst the remaining sites. A future version of the AMS will look at first attempting to find an alternative resource to host the SM before moving processes off site.

5 Computational Experiments

This section describes the experiments carried out to examine the EasyGrid AMS's ability to efficiently complete the execution of its application even in the presence of resource or process failures. Our focus is on determining the performance overheads associated with implementing an autonomic application based on the globus-enabled LAM/MPI. For the performance analysis, we consider a synthetic application representative of Bag-of-Task, Master-Worker, Parameter Sweep and SPMD applications – a class commonly executed on grids. In this test application (TApp), each process receives and sends a message to an application master process. This allows us to investigate how granularity and communications affect the performance of the EasyGrid AMS with respect to fault tolerance (FT). The experiments were carried out in a semi-controlled three-site grid environment interconnected by Gigabit and fast Ethernet switches. All the available resources run Linux Fedora Core 2, Globus Toolkit 2.4 and LAM/MPI 7.0.6. Sites 1, 2 and 3 are composed of Pentium IV 2.6 GHz processors with 512Mb RAM, where Site 1 contains 13 processors and Sites 2 and 3 have 7 and 5 processors, respectively. While the experiments were carried out with exclusive access to the computing resources, being connected on a public network meant that application performance could have been affected by external network traffic.

The first issue to be addressed is the overhead of the EasyGrid AMS implementation of TApp (executing under the 1PProc execution model) in comparison with a traditional LAM/MPI equivalent executing under the 1PTask model. Table 1 presents the percentage overhead (O) for the EasyGrid AMS TApp application with 500, 1000, 2000, 4000 and 8000 processes (P) and granularities (G) of 5, 10, 20, and 40 seconds. In the master-worker LAM/MPI version, the master process distributes units of workload equivalent to that of a process P to 24 worker processes on demand. While as expected the results show that the AMS execution is slightly slower than that of pure LAM/MPI for a homogeneous environment, this scenario is reversed in shared heterogenous Grids [4].

Next we look more closely at the degree of intrusion caused by the self-healing mechanism itself. The columns labelled Case 1 and Case 2 in Table 2 refer to the average execution times, in seconds, of TApp with two versions of the EasyGrid AMS (without and with the FT code (i.e., EasyGrid AMS of Table 1), respectively) in a fault free environment. The column adjacent to Case 2 presents the percentage overhead (O) for the AMS with fault tolerance. For a given number of

Table 1. Average execution times for the LAM/MPI and AMS implementations of the TApp application with varying numbers of processes and granularities

P	Granularity 5s			Granularity 10s			Granularity 20s			Granularity 40s		
	LAM	AMS	O(%)	LAM	AMS	O(%)	LAM	AMS	O(%)	LAM	AMS	O(%)
500	105.25	127.18	20.84	209.79	231.22	10.22	418.88	454.00	8.39	836.99	897.55	7.24
1000	209.59	229.31	9.41	418.49	440.51	5.26	836.17	878.86	5.11	1671.58	1748.57	4.61
2000	418.47	449.80	7.49	836.16	863.18	3.23	1678.49	1719.37	2.44	3342.40	3429.97	2.62
4000	831.25	892.33	7.35	1661.70	1710.47	2.93	3322.64	3411.79	2.68	6644.59	6814.09	2.55
8000	1661.69	1833.19	10.32	3322.50	3411.71	2.69	6690.61	6791.23	1.50	13287.94	13561.43	2.06

Table 2. Average execution times for the TApp applications with varying numbers of processes and granularities under differing scenarios: Case 1 - AMS without FT code; Case 2 - AMS with FT executed without failures; Case 3 - Hardware failures on 6 HM machines; Case 4 - Software failures in 6 HM processes; Case 5 - Hardware failure in Site 1's SM machine; Case 6 - Software failure in the SM process. One resource executes the GM and application master process, the remaining 24 resources each execute a HM and the application processes P. One resource at each site also executes a SM process.

G	P	Case 1	Case 2	O(%)	Case 3	O(%)	Case 4	O(%)	Case 5	O(%)	Case 6	O(%)
5	500	107.96	127.18	17.80	168.48	35.89	131.89	22.16	189.52	19.21	135.65	25.65
	1000	215.71	229.31	6.31	295.24	19.12	255.29	18.35	350.22	10.18	257.06	19.17
	2000	425.20	449.80	5.78	550.08	11.67	456.73	7.42	672.68	6.34	474.47	11.59
	4000	846.04	892.33	5.47	1034.97	5.28	905.42	7.02	1325.32	5.35	910.41	7.61
	8000	1686.79	1833.19	8.68	2063.06	5.34	1872.60	11.02	2654.99	5.63	2056.87	21.94
10	500	218.16	231.22	5.99	291.89	17.19	241.83	10.85	372.59	16.77	248.62	13.96
	1000	426.99	440.51	3.17	539.13	9.25	454.53	6.45	691.11	9.10	469.29	9.91
	2000	846.35	863.18	1.99	1020.59	3.81	879.47	3.91	1327.94	5.13	883.06	4.34
	4000	1689.58	1710.47	1.24	1997.45	1.66	1726.39	2.18	2604.79	3.58	1729.75	2.38
	8000	3372.26	3411.71	1.17	3982.15	1.69	3426.19	1.60	5120.59	1.88	3428.95	1.68
20	500	439.27	454.00	3.35	544.16	8.91	473.91	7.88	705.20	10.25	482.46	9.83
	1000	853.37	878.86	2.99	1029.30	4.32	898.02	5.23	1369.80	8.14	906.49	6.22
	2000	1695.52	1719.37	1.41	2024.58	2.89	1746.47	3.00	2600.77	2.89	1763.39	4.00
	4000	3380.69	3411.79	0.92	3993.79	1.61	3436.36	1.65	5129.99	1.98	3445.70	1.92
	8000	6744.91	6791.23	0.69	7945.10	1.44	6819.75	1.11	10169.30	1.16	6821.11	1.13
40	500	871.62	897.55	2.97	1047.49	5.19	919.78	5.52	1380.10	8.17	951.87	9.21
	1000	1722.36	1748.57	1.52	2053.40	3.64	1777.21	3.18	2736.37	7.68	1806.06	4.86
	2000	3383.75	3429.97	1.37	4025.92	2.39	3480.69	2.86	5190.76	2.75	3496.43	3.33
	4000	6754.70	6814.09	0.88	7963.06	1.35	6861.38	1.58	10239.25	1.81	6883.20	1.90
	8000	13486.71	13561.43	0.55	15748.25	0.54	13594.67	0.80	20659.57	2.77	13622.61	1.01

processes, the FT monitoring overhead falls as the process granularity increases since the execution is longer but the number of messages remains the same. However for a given granularity, as the number of processes in the application is increased so the overhead also decreases instead of remaining constant as one would expect. This due to the AMS's ability to reallocate application processes to overcome load imbalances caused by the FT subsystem.

Two aspects need to be evaluated with regard to the degree of intrusion when recovering from fail-stop faults. The cost of redistributing processes among other management processes when a hardware fault occurs and the cost of recreating the management and/or application processes in the case of software faults. While in Case 3, 25% (6) of the resources which run HMs fail, in Case 4, 25% of the HM processes fail (the lamd daemons remain active permitting HMs to be recreated). In Cases 5 and 6, the previous scenarios are repeated but this time the SM in charge of half (12) of the computational resources suffers a hardware and software fault respectively. The faults occur approximately at the midpoint of the application's corresponding Case 2 execution. The overheads are relative to Case 1 and, in Cases 3 and 5, take into consideration the loss in available computing power. With exception of the executions of TApp for 500 5s processes, the overheads for recovery are extremely small given the comparison to the equivalent program without FT code. As before, these overheads shrink to less than 1% as the number of processes and granularity increases. An interesting observation can be made with respect to the execution model. In Case 1, the

best execution times for a given total workload were achieved by programs with process granularities of 5s. In the remaining Cases, the best times are generally still achieved by the programs with small granularies (10s), even though the smallest FT overheads were obtained for larger granularities. The reason is the performance tradeoffs between the scheduling and fault tolerance mechanisms.

6 Conclusions and Future Work

The EasyGrid Application Management System is a middleware tailored to transparently grid-enable MPI applications. This paper discussed our experience with standard LAM/MPI and presented a low intrusion application-level strategy for implementing self-healing in autonomic MPI applications based on the EasyGrid AMS. Fault tolerance strategy is integrated with the AMS functions and takes advantage of the 1PTask execution model. The combination of a distributed message log (shared with process managers which hold messages for application processes that have yet to be created) and the re-start of failed processes can be effective. Evaluations are currently being carried out with AMS specific implementations (with support for commonly used collective communications) for real large scale scientific applications.

References

1. IBM Research: Autonomic computing. http://www.research.ibm.com/autonomic
2. Sterritt, R., Parashar, M., Tianfield, H., Unland, R.: A concise introduction to autonomic computing. Advanced Engineering Informatics 19(3), 181–187 (2005)
3. Nascimento, A.P., Sena, A.C., da Silva, J.A., Vianna, D.Q.C., Boeres, C., Rebello, V.: Managing the execution of large scale MPI applications on computational grids. In: Proc. of the 17th Symposium on Computer Architecture and High Performance Computing, Rio de Janeiro, Brazil, pp. 69–76. IEEE Computer Society Press, Los Alamitos (2005)
4. Sena, A.C., Nascimento, A.P., da Silva, J.A., Vianna, D.Q.C., Boeres, C., Rebello, V.: On the advantages of an alternative MPI execution model for grids. In: Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid, Rio de Janeiro, Brazil, IEEE Computer Society Press, Los Alamitos (2007)
5. Elnozahy, M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3), 375–408 (2002)
6. Foster, I.: Designing and Programming Parallel Programs. Addison-Wesley, Reading (1995)
7. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., Selikhov, A.: MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In: Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, pp. 1–18. IEEE Computer Society Press, Los Alamitos (2002)
8. Louca, S., Neophytou, N., Lachanas, A., Evripidou, P.: MPI-FT: Portable fault tolerance scheme for MPI. Parallel Processing Letters 10(4), 371–382 (2000)
9. Fagg, G.E., Dongarra, J.: Building and using a fault tolerant MPI implementation. Int. J. High Performance Applications and Supercomputing 18(3), 353–361 (2004)

Fault Tolerant File Models for MPI-IO Parallel File Systems

A. Calderón[1], F. García-Carballeira[1], Florin Isailă[1],
Rainer Keller[2], and Alexander Schulz[2]

[1] Computer Architecture Group, Computer Science Department
Universidad Carlos III de Madrid, Leganés, Madrid, Spain
acaldero@arcos.inf.uc3m.es
[2] High Performance Computing Center Stuttgart (HLRS),
Universität Stuttgart, 70550 Stuttgart, Germany
keller@hlrs.de

Abstract. Parallelism in file systems is obtained by using several independent server nodes supporting one or more secondary storage devices. This approach increases the performance and scalability of the system, but a fault in one single node can make the whole system fail. In order to avoid this problem, data must be stored using some kind of redundant technique, so that it can be recovered in case of failure. Fault tolerance can be provided in I/O systems by using replication or RAID based schemes. However, most of the current systems apply the same technique of fault tolerant at disk or file system level.

This paper[1] describes how fault tolerance support can be used by MPI applications based on *PVFS* version 2 [1], a well-know parallel file system for clusters. This support can be applied to other parallel file systems with many benefits: fault tolerance at file level, flexible definition of new fault tolerance scheme, and dynamic reconfiguration of the fault tolerance policy.

Keywords: Parallel File System, clusters, fault-tolerance, data declustering, reliability.

1 Introduction

For many parallel (and non-parallel) applications the performance of parallel I/O subsystem is critical. This is especially due to the fact that improvements in the CPU [2] performance are considerable larger than those of I/O components. The main improvement of the I/O subsystem performance comes from parallelism. However, on the other hand, the parallel I/O also is becoming critical for reliability reasons.

As it can be seen in the top 500 supercomputer list [3], the 72,2 % of current supercomputers are clusters. A large number of nodes with commercial off-the-shelf (COTS) hardware and software is used in order to build a supercomputer at a low cost. This approach improves the system performance and scalability, but increases the probability

[1] This work has been was carried out under the *HPC-EUROPA project* (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme. This work has also been partially supported by the Spanish Ministry of Science and Technology under the TIC2004-02156 contract.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 153–160, 2007.

of a single component failure. Parallel applications may scale, but are also very sensitive to errors: if a single node fails, the entire parallel application may fail as well.

For example, for systems like Blue Gene the main time to fail (MTTF) is less than 20 hours [4]. However, a parallel application typically is running for a longer period of time. Consequently, a fault tolerant mechanism is strongly needed. The most common solution is checkpointing [4]. However, the efficiency and correctness of checkpointing depends on the performance and reliability of the I/O subsystem (if the storage system fails, the checkpointing mechanism fails, too). Additionally, not all applications implement checkpointing.

Existing solutions for reliability are based on RAID [5] techniques, and on replication (one or more copies) at node level. Parallel file systems manage the fault tolerant support transparently to the user. In the current solutions, only the administrator can change the configuration of the parallel file system in a system-wide manner.

The main goal of our work is to give the applications the possibility of customizing the employed fault tolerant scheme. This includes the ability to define the fault tolerant policy per file rather than per file system. The user may choose to use or not the fault tolerant mechanism, may set up the number of I/O nodes involved, or may even implement a specialized fault-tolerant policy that matches best the application needs (in terms of reliability, in terms of performance by trying to match the application access patterns for example, etc.). This approach addresses the lack of any fault tolerance protection of the local disks of the compute nodes.

This article presents how these new ideas have been applied to PVFSv2 and can be used through MPI-IO [6]. We leverage our experience of implementing fault tolerant support in the *Expand* parallel file system [7]. This work presents how the fault tolerant support has been added into PVFSv2, and how MPI users could define a more flexible data block distribution (for data and for redundant information, too). Even more, it is possible to use the compute node to store data. If a compute node fails, the data stored on its local disk could be recovered.

The rest of this article is organized as follow. The Section 2 describes the modified architecture of PVFSv2. Section 3 shows the initial evaluation performed over the first prototype. Finally, section 5 summarizes and overviews the potential future work.

2 The Proposed PVFS Version 2 Architecture

The parallel applications run on compute nodes and send I/O requests though the PVFSv2 client library or kernel module. Other PVFSv2 components are the metadata and I/O servers that run on a multi-role server (one PVFSv2 server can act as an I/O node, as a metadata node or both).

The PVFS data distribution indicates how the user file data is distributed among PVFS I/O servers. The distribution maps the logical offset of the user file to the physical offset in the subfile on the PVFS I/O server. The associated distribution for a file, the name of the distribution and the services to be implemented are indicated on file creation, as a parameter of the *PVFS_sys_create* function. By default, PVFS uses a round-robin distribution of files over several I/O nodes. Each I/O node store the corresponding data in a subfile.

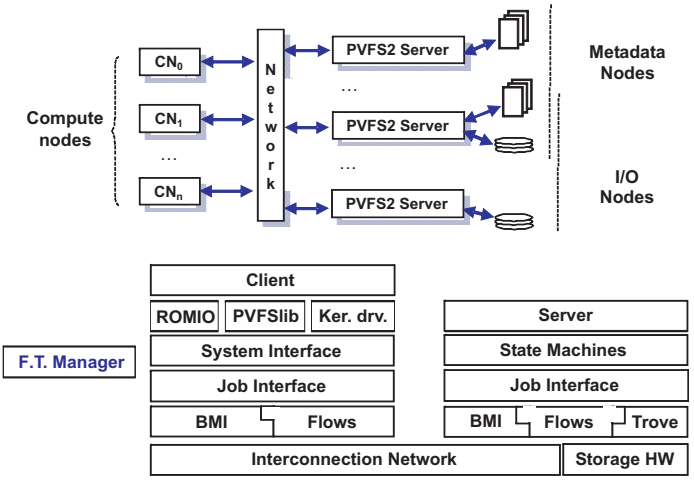


Fig. 1. PVFSv2 architecture: components and layers

At this moment, the fault tolerance has been implemented for data and no for metadata. Although there could be several metadata nodes, they are employed for load balancing purpose and not for fault tolerance.

Figure 1 shows the main layers and their main subcomponents. The novelty is a Fault-Tolerance (FT) Manager that is in charge of enforcing the fault tolerance policy. Our code intercepts the PVFSv2 requests, so that if the request is for a non-fault tolerant parallel file the existing PVFSv2 code is called. Otherwise, for a parallel file with fault tolerant support (or parallel files without the PVFSv2 default data block distribution), the F.T. manager module offers the new functionality by using the existing PVFSv2 code and some new algorithms.

In order to implement the fault tolerant support, we introduce a new PVFS data distribution, complementary to the default round-robin distribution of files. The data access services are provided by the F.T. Manager. Additionally, some extra attributes for describing the fault tolerant support were introduced. They are stored together with the regular attributes of PVFS.

The main attributes for describing the fault tolerant support are:

- stripe-size: size of the block used as unit of storage per server.
- server-count: number of PVFS servers over which the file is stored.
- server-status-list: list containing the status information of the servers used by this file.
- file redundancy scheme:
  - L_d vector: a list of tuples with the data blocks distribution. For each tuple $(x, y), \cdots, x$ is the server identifier and y is the subfile offset.
  - L_r vector: a list of tuples with the redundant blocks distribution. It is in the same format as the L_d vector.
  - L_{dr} vector: a list of triples describing the relationship between a data block and its associated redundant block(s). In the form of $(x, y, m), \cdots$, where x is the

server identification, y is the offset and m is the redundant method used. By now two methods are used: copy (letter 'C') and parity (letter 'P').

- L_{rd} vector: a list of triples with the relationship between a redundant block and its associated data block(s). It has the same format as the L_{dr} vector.

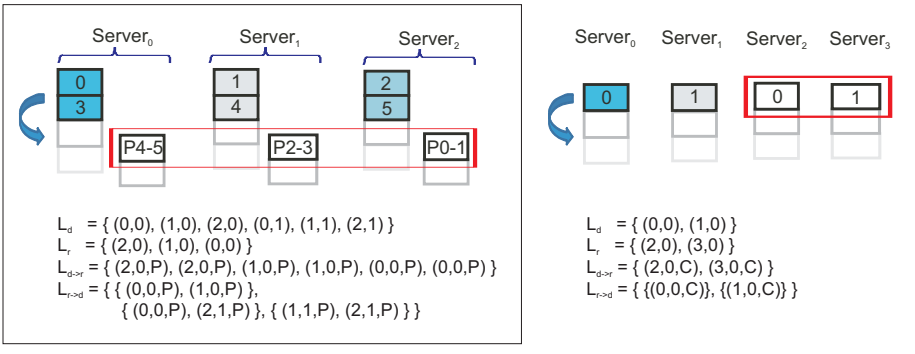


Fig. 2. Example of RAID 5 with outer redundant (left) and RAID 1 (right)

For fault tolerant parallel files, the former four vectors are necessary. For parallel files without the default PVFSv2 data block distribution, only the first vector is sufficient. As an example, Figure 2 shows a RAID 5 distribution, where the parity information is stored in different subfiles than user data information. In this way, the redundant information can be added or removed without rewriting the full parallel file. As the example shows, the distribution is a pattern that it is repeated throughout the file.

The F.T. manager module implements the algorithms needed to manage the normal behavior (without any fault) and the degradated mode (after a fault). In both cases, normal and degradated, the open, creat, close, read and write operations must take the most appropiate actions in order to work properly. If a fault is repaired, the F.T. manager module also takes care of the data recovery actions in order to update the modifications that have been done during the period of time the server has been down (roll-forward).

2.1 The Integration in MPI-IO: A Case of Study with PVFS Version 2

In MPI, a hint allows users to provide special information about a file An MPI hint is a variable-value pair. We propose the use of one variable in order to associate a file redundancy scheme to an MPI file. This hint will be passed to the PVFSv2 API, so that the redundancy scheme could be applied. Although we propose the use of PVFS, the mechanism could be used for other parallel file systems as well.

Figure 1.1 shows how this MPI hint is used to define a new redundancy scheme and how this is associated to a file. In this example the error checking has been omitted for clarity.

Put in simple way, the idea is to provide some extra functionality that let users select the most appropriate scheme from general properties (number of failures to be tolerated, useful space for data, etc.)

Listing 1.1 Example of how to use *MPI_Info_set* to define a fault tolerant scheme

```
MPI_File fh ;
MPI_Info info ;

...
MPI_Info_set(
          info,
          "SCHEME-SET-R5O",
          "R50 4
           Ld  12  0,0,    1,0,     2,0,    3,0,
                   0,1,    1,1,     2,1,    3,1,
                   0,2,    1,2,     2,2,    3,2
           Lr   4  3,0,    2,0,     1,0,    0,0
           Ldr 12  3,0,0,  3,0, 0,  3,0,0,  2,0,0,
                   2,0,0,  2,0, 0,  1,0,0,  1,0,0,
                   1,0,0,  0,0, 0,  0,0,0,  0,0,0
           Lrd 4   9,0,0,  6,0, 0,  3,0,0,  0,0,0"
      ) ;
...
MPI_File_open(MPI_COMM_WORLD, ''pvfs2:/pool01/f07.dat'',MPI_MODE_RDWR, info, &fh);
```

3 Evaluation

The evaluation platform was the Cacau supercomputer (see Figure 3). It has 200 dual Intel Xeon EM64T processors. a Peak Performance of 2.5 TFlops. All nodes are interconnected by an Infiniband network (1000 MB/sec.) and a Gigabyte network. Each node has 1, 2 and 6 GB of RAM. During the evaluation, nodes with 1 GB were used. Each node may access a scratch directory of 58GB. A global shared scratch of 960GB and a global shared home directory. justify why there are a global shared scratch

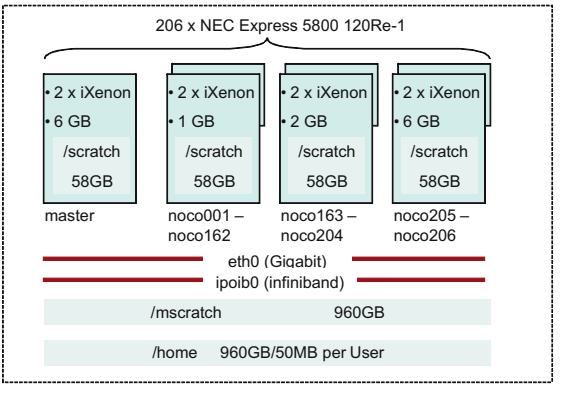


Fig. 3. The CACAU configuration on the evaluation

PVFSv2 is customizable by the user and can be installed on-demand, without administrator privileges. By using this capability we can build a dynamic fault tolerant partition over the local scratches of the individual nodes. In our experiments we have used Open MPI 1.1.1 [8].

3.1 The Evaluation Process: Benchmarking and Configuration

In a parallel file system, the larger the number of distributed components, the higher is the potential aggregate performance, but also the higher the probability of a fault of a single node. The key aspect to be evaluated is the impact of fault tolerant support on the performance of the parallel file system.

We use a simple benchmark: one process that writes (and reads) 2 GB of data to a single file over a PVFSv2 partition with 2, 4, and 8 servers, and varying the access size (from 8KByte to 1MByte). Three redundancy schemes have been compared: RAID 0 (the classic cyclic block distribution without fault tolerant support, that is the baseline), RAID 1 (as shown on the right part of Figure 2 for 4 servers) and RAID 5 (as shown on the left part of Figure 2 for 4 servers).

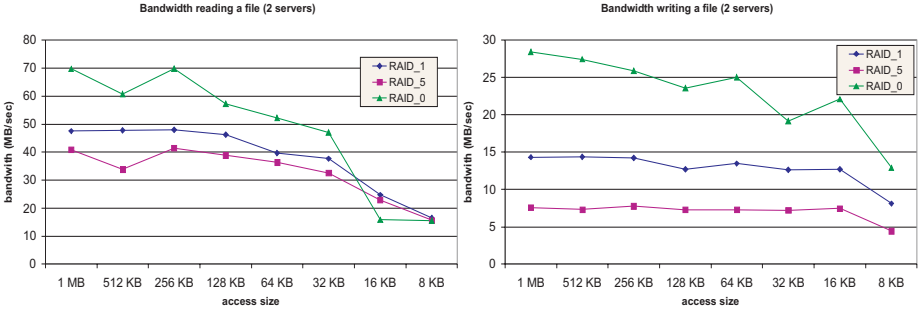


Fig. 4. Bandwidth while reading (left) and writing (right) with a PVFSv2 partition of 2 I/O nodes

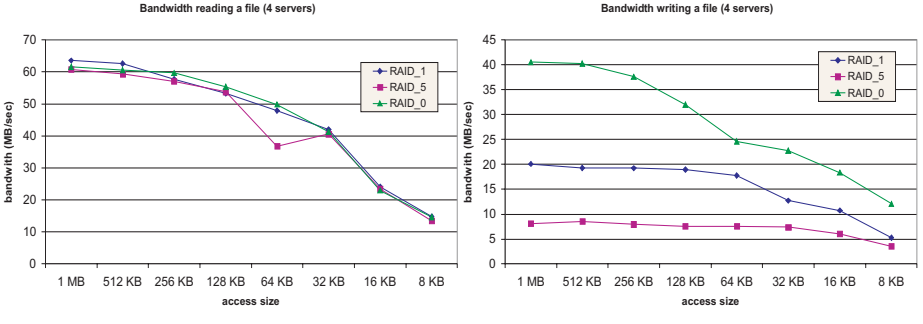


Fig. 5. Bandwidth while reading (left) and writing (right) with a PVFSv2 partition of 4 I/O nodes

Figures 4, 5, and 6 show the obtained results. The main conclusions drawn from the analysis of these results are the following:

- The performance is poor in fault tolerant schemes while updating the file because both data and redundant information must be updated
- In general, when the number of servers increases, the performance is better (more throughput)

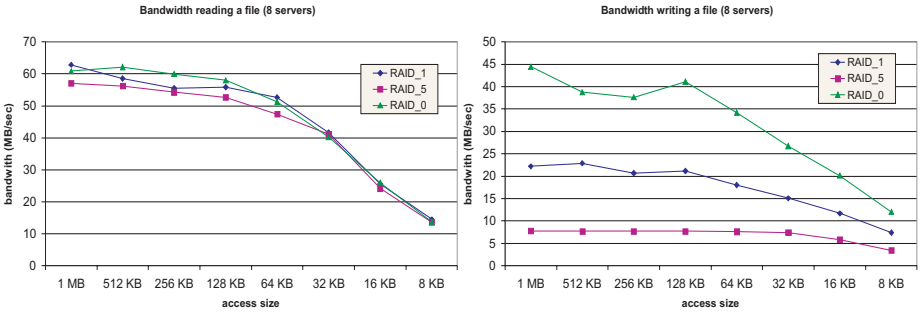


Fig. 6. Bandwidth while reading (left) and writing (right) with a PVFSv2 partition of 8 I/O nodes

– RAID 5 works in a transactional way at block level, so that when the number of servers increases, then the performance is almost the same.

The replication based solution scales better (RAID 1) for writing because several updates of blocks can be done in parallel first, then the system can update the replicated blocks. The RAID 5 was implemented in a conservative transactional way, by modifying one by one first the original block and if successful, the parity. The full parallelism could be potentially achieved from writes of independent parity groups. This is one of our future work plans.

4 Related Work

A well know solution to provide fault tolerant support to PVFS by using a RAID 10 scheme is CEFT-PVFS [9]. This solution is limited to only one redundancy scheme, and for all files in the partition. Another interesting approach is Clusterfile [10]. It doesn't offer fault tolerant support but introduce a configurable layout per file in the parallel file system. The most useful RAID schemes (specially for hard disk) is not limited to the RAID 0, RAID 1, and RAID 5. An example of combination of ideas beyond traditional RAID schemes is RAID-x [11]. This (and other schemes) could be also used in parallel file systems at file level.

The *Expand* parallel file system [7] was the first parallel file system in which flexible redundant schemes could be used at file level.

5 Conclusions and Future Work

In this paper we have presented a new mechanism for defining redundancy schemes for files in MPI-IO. Each redundancy scheme may be applied at file level and relays on the underlaying parallel file system support. This support was implemented in a prototype version in PVFSv2.

Our main future work plans include: adding fault tolerance to metadata servers, optimize the RAID 5 algorithm (the initial implementation has not parallelism in large block updates), more evaluation of redundancy schemes under various environment settings, integrate the support for fault tolerance into other parallel file systems.

References

1. Miller, N., Latham, R., Ross, R., Carns, P.: improving cluster performance with pvfs2. ClusterWorld Magazine 2(4) (2004)
2. Moore, G.E.: Cramming more components onto integrated circuits. Electronics 38(8) (1965)
3. 500, T.: The top 500 supercomputer list (2006)
4. da Lu, C.: Scalable diskless checkpointing for large parallel systems. University ofIllinois at Urbana-Champaign (2005)
5. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (RAID). In: Boral, H., Larson, P.A. (eds.) Proceedings of 1988 SIGMOD International Conference on Management of Data, Chicago, IL, pp. 109–116 (1988)
6. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the 1999 IOPADS, pp. 23–32 (1999)
7. Calderon, A., Garcia-Carballeira, F., Carretero, J., Perez, J.M., Sanchez, L.M.: A fault tolerant mpi-io implementation using the expand parallel file system. In: PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05, pp. 274–281. IEEE Computer Society Press, Washington, DC, USA (2005)
8. Gabriel, E., Graham, R.L., Castain, R.H., Daniel, D.J., Woodall, T.S., Sukalski, M.W., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barett, B., Lumsdaine, A.: Open MPI: Goals, concept, and design of a next generation MPI. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Springer Verlag, Heidelberg (2004)
9. Zhu, Y., Jiang, H., Qin, X., Feng, D., Swanson, D.R.: Improved read performance in a cost-effective, fault-tolerant parallel virtual file system (CEFT-PVFS). In: Workshop on Parallel I/O in Cluster Computing and Computational Grids, Tokyo, IEEE Computer Society Press (May 2003) 730–735 Organized at the IEEE/ACM International Symposium on Cluster Computing and the Grid (2003)
10. Isaila, F., Tichy, W.F.: Clusterfile: A flexible physical layout parallel file system. Concurrency and Computation: Practice and Experience 15(7-8), 653–679 (2003)
11. Hwang, K., Jin, H., Ho, R.: RAID-x: A new distributed disk array for I/O-centric cluster computing. In: Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, PA, pp. 279–287. IEEE Computer Society Press, Los Alamitos (2000)

An Evaluation of Open MPI's Matching Transport Layer on the Cray XT

Richard L. Graham[1], Ron Brightwell[2], Brian Barrett[3], George Bosilca[4], and Jelena Pješivac-Grbović[4]

[1] Oak Ridge National Laboratory*
Oak Ridge, TN USA
rlgraham@ornl.gov
[2] Sandia National Laboratories**,
Albuquerque, NM USA
rbbrigh@sandia.gov
[3] Los Alamos National Laboratory***
Los Alamos, NM USA
bbarrett@lanl.gov
[4] The University of Tennessee,
Knoxville, TN USA
{bosilca,pjesa}@cs.utk.edu

Abstract. Open MPI was initially designed to support a wide variety of high-performance networks and network programming interfaces. Recently, Open MPI was enhanced to support networks that have full support for MPI matching semantics. Previous Open MPI efforts focused on networks that require the MPI library to manage message matching, which is sub-optimal for some networks that inherently support matching. We describes a new matching transport layer in Open MPI, present results of micro-benchmarks and several applications on the Cray XT platform, and compare performance of the new and the existing transport layers, as well as the vendor-supplied implementation of MPI.

1 Introduction

The Open MPI implementation of MPI is the result of an active international open-source collaboration between industry, research laboratories, and academia. In a short time, Open MPI has evolved into a robust, scalable, high-performance implementation for a wide variety of architectures and interconnects. It is

* Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

** Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

*** Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 161–169, 2007.

currently being run in production on several of the largest production computing systems in the world. Much of the current effort in developing Open MPI has targeted networks and network programming interfaces that do not support MPI matching semantics. These networks depend on the MPI implementation to perform message selection inside the MPI library. As such, existing transport layers in Open MPI were designed to provide this fundamental capability. Unfortunately, these transport layers have been shown to be sub-optimal in some cases for networks that support MPI matching semantics, mostly due to redundant functionality.

Recently, a new transport layer has been developed that is designed specifically for networks that provide MPI matching semantics. This new transport layer eliminates much of the overhead of previous transport layers and exploits the capabilities of the underlying network layer to its fullest. This paper describes this new matching transport layer and its implementation on the Cray XT platform. We compare and contrast features of the new transport with the existing non-matching transport layer. Performance results from several micro-benchmarks demonstrate the capabilities of the new transport layer, and we also show results from several real-world applications. We also include performance results for the native vendor-supplied MPI implementation.

The rest of this paper is organized as follows. Section 2 presents an overview of the Open MPI implementation for the Cray XT platform, the Cray MPI implementation, and the test platform for experiments presented in this paper. Results for microbenchmarks and applications are presented in Sections 3 and 4, respectively. Relevant conclusions are presented in Section 5.

2 Background

The Cray XT4 platform utilizes the Portals [1] interface for scalable, high performance communication. Portals provides a number of features not common to high performance networks, particularly rich receive matching capable of implementing the MPI message matching rules. Initial work with Open MPI on the XT4 treated Portals like traditional commodity networks [2]. Recent work extends Open MPI to take advantage of Portals' rich feature set.

2.1 Open MPI Point-to-Point Architecture

Open MPI implements point-to-point MPI semantics utilizing a component interface, the Point-To-Point Management Layer (PML) [3]. The PML is responsible for implementing all MPI point-to-point semantics, including message buffering, message matching, and scheduling message transfers. The general architecture is shown in Figure 1. At run-time, one PML component will be selected and used for all point-to-point communication. Three PMLs are currently available: OB1, DR, and CM.[1] The PMLs can be grouped into two categories based based on responsibility for data transfer and message matching: OB1 and DR or CM.

[1] PML names are internal code names and do not have any meaning.

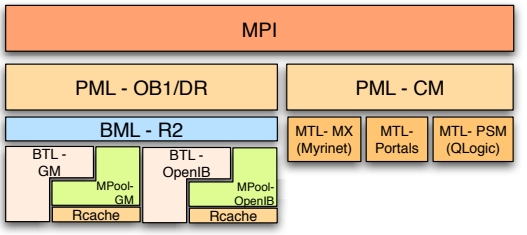


Fig. 1. Open MPI's Layered Architecture

OB1 and DR. OB1 and DR implement message matching and data transfer scheduling within the MPI layer, utilizing the BTL interface for data transfer. OB1 provides high performance communication on a variety of HPC networks and is capable of utilizing remote direct memory access (RDMA) features provided by the underlying network. DR is focused on data integrity and only utilizes send/receive semantics for message transfer. Both PMLs share the lower level Byte Transfer Layer (BTL), which is the layer that handles the data transfer, the supporting Byte Management Layer (BML), Memory Pool (MPool), and the Registration Cache (Rcache) frameworks. While these are illustrated and defined as layers, critical send/receive paths bypass the BML, which is used primarily during initialization and BTL selection.

When using OB1 and the Portals BTL, short messages are sent eagerly and long messages are sent using a rendezvous protocol. Eager message transfer involves a copy into BTL-specific buffers at the sender and a copy out of BTL-specific buffers at the receiver. For long messages, a Portals RDMA get is issued to complete data transfer directly into the application receive buffer. User-level flow control ensures messages are not dropped, even for large numbers of unexpected sends.

CM. The CM PML provides request management and handling of buffered sends, relying on the MTL framework to provide message matching and data transfer. The MTL is designed specifically for networks such as Portals or Myrinet MX, which are capable of implementing message matching inside the communication library. Unlike OB1, which supports multiple simultaneous BTLs, only one MTL may be utilized per application.

The Portals MTL utilizes a design similar to that described in [1]. The Portals MTL sends all data eagerly, directly from application buffers. If a receive has been pre-posted, the data is delivered directly to the user buffer. Unexpected short message, less than 32K currently, are buffered in MTL level buffers. Unexpected long messages are truncated, and after a match is made Portal's RDMA get functionality completes the data transfer. With the exception of unexpected receives, messages are matching by the Portals library. The Portals MTL is designed to provide optimal performance for applications that pre-post their receives.

OB1 and CM fundamentally differ in the handling of long messages. The OB1 protocol uses a rendezvous protocol with an eager limit of 32K bytes. On the

receive side the memory descriptors are configured to buffer this data if messages are unexpected. For large messages, the OB1 protocol attempts to keep network congestion down, so sends only a header used for matching purposes. Once the match is made, the Portals get method is used to deliver the user's data in a zero copy mode, if the MPI data type is contiguous, directly to the destination. This mode of point-to-point communications is very useful when an application run uses a lot of large unexpected messages, i.e. when the message is sent to the destination, before the receive side has posted a matching receive.

CM does not specify a protocol for long messages, leaving such decisions to the MTL. The Portals MTL procotol is agressive on sending data. Both the short and the long protocol send all user data at once. If there is a matching receive posted, the data is delivered directly to the user destination. In the absence of such a posted receive, short messages, i.e. messages shorter than 32K bytes, are buffered by the receive Portals memory descriptor. However, all he data associated with long messages is dropped, and a Portals get request is performed after the match is made to obtain the data. This protocol is aimed at providing the highest bandwidth possible for the application.

2.2 Cray MPI

Cray MPI is derived from MPICH-2 [4], and supports the full MPI-2 standard, with the exception of MPI process spawning. This is the MPI implementation shipped with the Cray Message Passing Toolkit. The communication protocol used by Cray MPI is generally similar to that of the Portals MTL, although there are significant differences regarding the handling of event queue polling.

2.3 Application Codes

Four applications, VH-1, GTC, the Parallel Ocean Program (POP), and S3D, were used to compare the protocols available on the Cray XT platform. VH-1 [5] is a multidimensional ideal compressible hydrodynamics code. The Gyrokinetic Toroidal Code [6] (GTC) uses first-principles kinetic simulation of the electrostatic ion temperature gradient (ITG) turbulence in a reactor-scale fusion plasma to study turbulent transport in burning plasmas. POP [7] is the ocean model component of the Community Climate System Model, which is used to provide input to the Intergovernmental Panel on Climate Change assessment. S3D [8] is used for direct numerical simulations of turbulent combustion by solving the reactive Navier-Stokes equations on a rectilinear grid.

2.4 Test Platforms

Application performance results were gathered on Jaguar, a Cray XT4 system at Oak Ridge National Laboratory. Jaguar is composed of 11,508 dual-socket 2.6 GHz dual-core AMD Opterons, and the network is a 3-D torus with the Cray SeaStar 2.1 [9] network. Micro-benchmark results were gathered on Red Storm, a Cray XT3+ system at Sandia National Laboratories. Red Storm contains 13,272 single-socket dual-core 2.4 GHz AMD Opteron chips, a SeaStar

2.1 network, which is torus in only one direction. The major difference between these two systems is the speed of the processor, and the communication micro-benchmarks can be scaled appropriately. For both systems, compute nodes run the Catamount lightweight kernel, and all network communications use the Portals 3.3 programming interface [10].

For the application results, the default Cray MPI installation, XT/MPT version 1.5.31 with default settings, is used for the benchmark runs. The trunk version of Open MPI (1.3 pre-release) is used for these runs, with data collected using both the Portal ports of the CM and OB1 PMLs. Open MPI's tuned collectives are used for collective operations. To minimize differences in timings due to processor allocations, all runs for a given application and processor count are sequentially run within a single resource allocation.

3 Micro-Benchmark Performance

We use several communication micro-benchmarks to compare the performance of the two MPI implementations. We first compare latency and bandwidth performance using the NetPIPE [11] benchmark. Figure 2(a) shows half round-trip ping-pong latency results. The Cray implementation has the lowest zero-length latency at 4.78 μs, followed by 4.91 μs and 6.16 μs respectively for Open MPI's CM and OB1. Figure 2(b) plots bandwidth performance. Results shows that beyond a message length of 100 bytes, Open MPI's CM bandwidth is higher than that of Cray MPI's, but eventually the curves join as the asymptotic peak bandwidth is reached. However, Cray MPI's bandwidth curve is consistently higher than that of Open MPI's OB1 protocol. Note also that Open MPI's transition from the short-message protocol to the long-message protocol produces a much smoother curve than Cray MPI's.

Next, we measure CPU availability for sending and receiving using the Sandia overhead benchmark [12]. This benchmark measures the percentage of the processor that is available to the application process while sending and

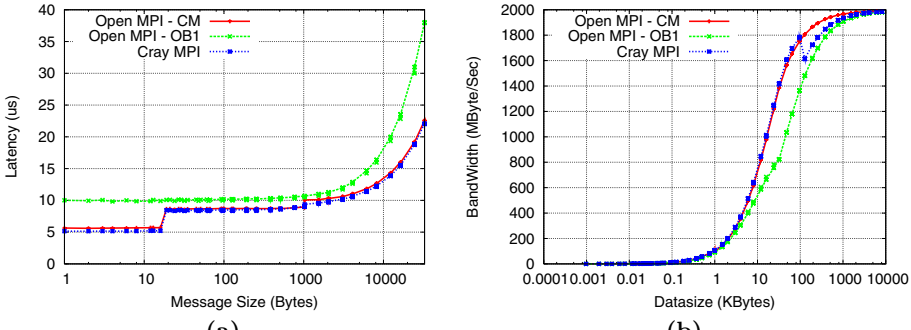


Fig. 2. NetPIPE (a) latency and (b) bandwidth performance

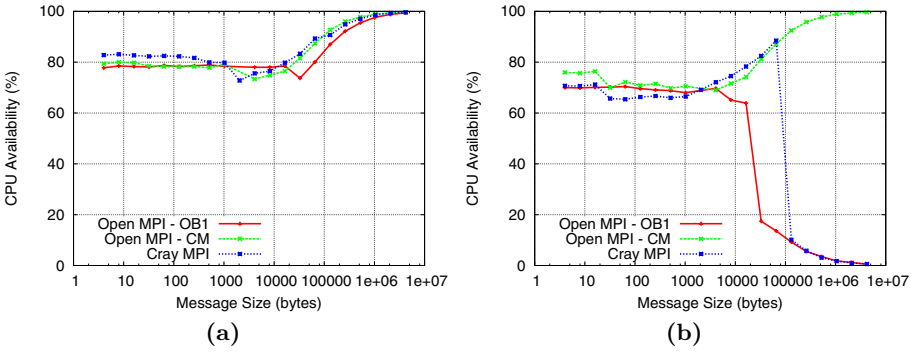


Fig. 3. SMB send availability (a) and receive availability (b)

receiving messages. Figure 3(a) shows send-side CPU availability, while Figure 3(b) shows receive-side CPU availability. On the send side, Cray MPI has a very slight advantage for very small messages sizes. However, for message sizes between 1 KB and 10 KB, the OB1 transport has a slight advantage over the other two. This is likely due to memory copies in Cray MPI and CM that reduce latency at the expense of CPU availability. Results for receive availability are much different. The CM transport has a slight advantage at small message sizes, but is able to maintain high availability for very large messages. The eager protocol messages in CM allow for nearly complete overlap of computation and communication. The other two curves show a rapid decrease in availability at the point where the eager protocol switches to a rendezvous protocol. Cray recently modified their implementation to use a rendezvous protocol by default, in spite of previous results that demonstrated high receive-side availability similar to CM.

For our last communication micro-benchmark, we examine message rate using a modified version of the Ohio State University streaming bandwidth benchmark. This benchmark measures the number of messages per second that can

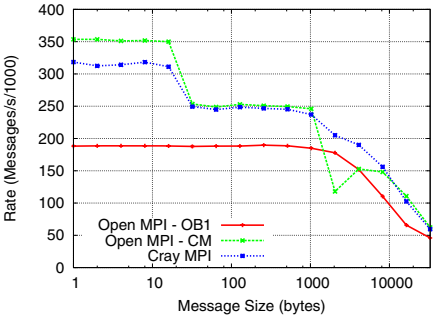


Fig. 4. Small message rate

be processed by streaming messages. In Figure 4 we can see that CM has an advantage over Cray MPI for message sizes up to about 32 bytes, at which point the curves almost converge. Performance of the CM transport drops significantly at 2048 bytes. The OB1 message rate is nearly half of the other implementations, due to both protocol overhead and rate limiting to ensure message reliability.

4 Application Performance

We compare the performance of VH-1, GTC, POP, and S3D, at medium process count. Figure 5 shows overall application run-time for these codes, with data for VH-1 and POP collected at 256 processes, and for GTC and S3D at 1024 processes. Overall, Open MPI CM PML slightly out-performs Cray MPI, and the CM PML consistently outperforming the OB1 PML.

VH-1 was run using 256 MPI processes, with the CM run-times being about 0.4% faster than the Cray MPI run-times, and about 0.3% faster than the OB1 runs. For the GTC runs at 1024 processes, the Cray MPI application run-times are about 4% faster than the Open MPI CM runs, and 15% than the OB1 runs. Running POP at 256 processes, Open MPI CM outperforms Cray MPI by about 3%, and outperforms CM by 18%. Finally, at 1024 processes, Open MPI's CM outperforms Cray MPI by 12%, and it outperforms OB1 by 3%.

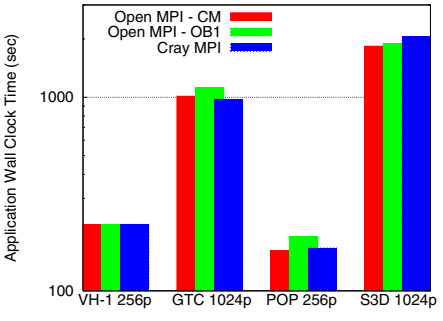


Fig. 5. Application Wall Clock Run-Time(sec)

Table 1 lists the fraction of run-time spent inside the MPI library, along with the most time consuming MPI functions. The data was collected using mpiP [13], with the CM PML. The average amount of time spent in MPI routines differs considerably from application to application, with 6.1% of GTC's run time at 1024 processes being spent in the MPI library, to 65.7% of POP's run-time at being spent in the MPI library. 7.9% of S3D's run-time and 16.9% of VH-1's run-time are spent in the MPI library. For applications other than S3D—which uses collectives sparingly—collective communications dominate the MPI traffic at large processor counts. POP spends 40.8% of the run-time performing small (8 byte) reduction operations. The collective communications used by Open MPI

Table 1. Application Communications Profile with Open MPI's CM Point-To-Point communications

App	# Procs	Ave MPI Time % min,max	Message Profile # short % total	# long % total	#dropped % long % min,max	Top MPI Routines %Tot time	%Tot time	%Tot time
VH-1	256	16.9% 15.5, 24.7	240 7.4	3000 92.6	890 29.6 9.1, 51.7	Alltoall 15.9	Allreduce 1.0	
GTC	1024	6.1% 2.9, 13.9	6524 43.7	8404 56.3	2130 25.3 7.6, 56.0	Allreduce 4.6	Sendrecv 1.3	Bcast 0.1
POP	256	65.7% 60.6, 70.5	5472986 99.9	5648 0.1	789 13.3 1.8, 93.5	Allreduce 40.8	Waitall 14.4	Isend 5.5
S3D	1024	7.9% 5.5, 9.1	946 0.4	225015 99.6	104020 46.2 25.1, 96.0	Wait 7.2	Allreduce 0.3	Barrier 0.2

use PML level communications for data exchange, and as such the performance of the Point-To-Point communications is one of the factors contributing to overall collective performance.

In addition, Table 1 lists the breakdown of Point-To-Point traffic for the applications. We categorize the data based the communication protocol used; either the short-message protocol used at or below 32K byte cutoff length or the long-message protocol. On average, S3D's, VH-1's, and GTC's Point-To-Point communications are dominated by long messages, with 99.6% of S3D's messages, 92.6% of VH-1's messages, and 56.3% of GTC's messages being long-messages. 46.2%, 29.6%, and 25.3% of the long-messages sent by these respective applications are dropped, and retransmitted once a match is made. While additional time is consumed retreiving the long-message data after the match is made, there does not appear to be a strong correlation between the fraction of long-messages being retransmitted and the overall application performance relative to the CM PML. POP communications are dominated by short-messages, and the long-message protocol is largely irrelevant to its performance in the current set of runs.

5 Conclusions

This paper compares the performance of the Point-To-Point performance of Open MPI's new CM PML with the OB1 PML and with Cray MPI utilizing the Portals communications library. Both micro-benchmarks and full application benchmarks are used. The CM PML is designed to make optimal use of Portals capabilities for providing good application performance at large scale. It provides message injection rates that are comparable to those of Cray MPI and consistently better than those obtained with the OB1 PML. It is superior to both Cray MPI and OB1 with respect to CPU availability, allowing nearly all of the CPU to be available during large message transfers on both the sender and the receiver. CM also has good latency and bandwidth performance curves, comparable with Cray MPI, but superior to the OB1 implementation. With regard to application performance, CM gives slightly better overall performance when compared with Cray MPI, and consistently better performance with respect to OB1.

References

[1] Brightwell, R., Maccabe, A.B.R.R.: Design, implementation, and performance of mpi on portals 3.0. International Journal of High Performance Computing Applications 17(1) (2003)
[2] Barrett, B.W., Brightwell, R., Squyres, J.M., Lumsdaine, A.: Implementation of open mpi on the cray xt3. In: 46th CUG Conference, CUG Summit 2006 (2006)
[3] Graham, R.L., Barrett, B.W., Shipman, G.M., Woodall, T.S., Bosilca, G.: Open mpi: A high performance, flexible implementation of mpi point-to-point communications. Parallel Processing Letters 17(1), 79–88 (2007)
[4] Argonne National Lab.: MPICH2. (http://www-unix.mcs.anl.gov/mpi/mpich2/)
[5] Blondin, J.M., Lufkin, E.A.: The piecewise-parabolic method in curvilinear coordinates. The Astorphysical Journal 88, 589–594 (1993)
[6] Lin, Z., Hahm, T.S., Lee, W.W., Tang, W.M., White, R.B.: Turbulent transport reduction by zonal flows: Massively parallel simulations. Science 281, 1835 (1998)
[7] Dukowicz, J.K., Smith, R., Malone, R.: A reformulation and implementation of the bryan-cox-semter ocean model on the connection machine. J. Atmospheric and Oceanic Tech. 10, 195–208 (1993)
[8] Hawkes, E., Sankaran, R., Sutherland, J., Chen, J.: Direct numerical simulation of turbulent combustion: Fundamental insights towards predictive models. Journal of Physics: Conference Series 16, 65–79 (2005)
[9] Alverson, R.: Red storm. Invited Talk, Hot Chips 15 (2003)
[10] Riesen, R., Brightwell, R., Pedretti, K., Maccabe, A.B., Hudson, T.: The Portals 3.3 Message Passing Interface - Revision 2.1. Technical Report SAND20006-0420, Sandia National Laboratory (2006)
[11] Snell, Q., Mikler, A., Gustafson, J.: In: IASTED International Conference on Intelligent Information Management and Systems (1996)
[12] Doerfler, D., Brightwell, R.: Measuring MPI send and receive overhead and application availability in high performance network interfaces. In: 13th European PVM/MPI Users' Group Meeting, Bonn, Germany (2006)
[13] mpiP: Lightweight, Scalable MPI Profiling http://mpip.sourceforge.net

Improving Reactivity and Communication Overlap in MPI Using a Generic I/O Manager

François Trahay, Alexandre Denis, Olivier Aumage, and Raymond Namyst

INRIA, LaBRI, Université Bordeaux 1
351, cours de la Libération
F-33405 TALENCE, France
{trahay,denis,aumage,namyst}@labri.fr

Abstract. MPI applications may waste thousands of CPU cycles if they do not efficiently overlap communications and computation. In this paper, we present a generic and portable I/O manager that is able to make communication progress asynchronously using tasklets. It chooses automatically the most appropriate communication method, depending on the context: multi-threaded application or not, SMP machine or not. We have implemented and evaluated our I/O manager with Mad-MPI, our own MPI implementation, and compared it to other existing MPI implementations regarding the ability to efficiently overlap communication and computation.

Keywords: Polling, Interrupt, Thread, Scheduler, High-Speed Network.

1 Introduction

Asynchronism is becoming ubiquitous in modern communication runtimes. This evolution is the combined result of multiple factors. *Firstly*, communication subsystems implement increasingly complex optimizations in order to make better use of networking hardware. As we have shown in [1], such optimizations require online analysis of the communication schemes and hence require the desynchronization of the communication request submission from its processing. *Moreover*, providing rich functionality such as communication flow multiplexing or transparent multi-method, heterogeneous networking implies that the runtime system should again take an active part in-between the communication request submit and processing. And *finally*, overlapping communication with computation and being reactive actually do matter more now than it has ever done [2,3]. The latency of network transactions is in the order of magnitude of several thousands CPU cycles at least. Everything must therefore be done to avoid independent computations to be blocked by an ongoing network transaction. This is even more true with the increasingly dense SMP, multicore, SMT (also known as Intel's Hyperthreading) architectures where many computing units share a few NICs.

Since portability is one of the most important requirements for communication runtime systems, the usual approach to implement asynchronous processing is

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 170–177, 2007.

to use threads (such as Posix threads). Popular communication runtimes indeed are starting to make use of threads internally and also allow applications to be multithreaded as it can be seen with both MPICH-2 [4], and Open MPI [5,6]. Low level communication libraries such as Quadrics' Elan [7] and Myricom's MX [8] also make use of multithreading. Such an introduction of threads inside communication subsystems is not going without troubles however. The fact that multithreading is still usually optional with these runtimes is symptomatic of the difficulty to get the benefits of multithreading in the context of networking without suffering from the potential drawbacks.

In this paper, we analyze the two fundamental approaches of integrating multithreading and communications —interrupts and polling. We study their respective benefits and their potential drawbacks, and we discuss the importance of the cooperation between the asynchronous event management code and the thread scheduling code in order to avoid such disadvantages. We then introduce our proposal for symbiotically combining both approaches inside a new generic network I/O event manager. The paper is organized as follows. Section 2 exposes the problem of integrating threads and communications. Section 3 introduces our proposal for a new asynchronous event management model and gives details about our implementation. We evaluate this implementation in Section 4 and Section 5 concludes and gives an insight of ongoing and future work.

2 Integrating Threads and Communication: The Problems of Network I/O Events Management

The detection of network I/O events can be achieved by two main strategies. The most common approach consists in using the *active waiting*: a polling function is called repeatedly until a network I/O event is detected. The polling function is usually inexpensive, but repeating this operation thousands of times may be prohibitive. The other method for detecting communication events is the *passive waiting* which is based on blocking calls. In that case, the NIC informs the operating system that a network I/O event has occurred by using an interrupt, making this method much more reactive than polling. However this operation involves interrupt handlers and context switches which are rather costly.

The best method to use depends on the application, but in both cases, some behaviors may lead to suboptimal performance. When using interrupt-based methods, priority issues may occur: the thread that is waiting for the communication event may be scheduled with some delay. This is the case when, for example, it has been computing for a long period before it blocks, lowering its priority. Moreover, the system has to support methods to detect the network I/O events. For instance, in a pure user-level scheduler, interrupt-driven blocking calls are prohibited (unless a specific OS extension like the *Scheduler Activations* [9] is used).

Using polling methods can also be problematic: if the system is overloaded (i.e. there are more running threads than available CPUs), the polling thread may scarcely be scheduled, thus increasing the reaction time. Moreover, some

asynchronous communication operations need a regular polling in order to progress. For instance, a *rendez-vous* requires a regular polling so that the preliminary phase makes progress. As it is shown in [2], some applications would significantly improve their execution time by efficiently overlapping communication and computation, which requires to poll communication events regularly.

3 An I/O Manager Model

To resolve these kinds of problems, we propose an I/O manager that provides the communication runtime systems with a network event detection service. Thus, communication libraries themselves become independent of the multithread issues and related hardware issues such as the number of CPUs. Thereby, they can focus their efforts on communication optimizations and other functionalities. By working closely with a specific thread scheduler, the I/O manager can be viewed as a progression engine able to schedule a communicating thread when needed or to dynamically adapt the polling frequency to maximize the reactivity/overhead ratio. The I/O manager handles both polling and interrupt-based methods, switching from one method to another depending on the context.

The implementation of our I/O manager called PIOMan (PM2 I/O Manager) relies on a two-level thread scheduler [10] which was slightly modified to interact with the I/O manager when necessary. The use of a two-level scheduler allows to precisely control thread scheduling at the user level, with almost no explicit (and expensive) interaction with the OS. This way, we can dynamically favour the scheduling of a thread requiring a high reactivity to communication events during a fixed period. PIOMan is available as three main versions: no-thread, mono (user-level threads) or SMP (user threads on top of kernel threads).

3.1 Overview of the I/O Manager

The mechanism of our I/O manager is described through an example shown in Figure 1: the application first registers a callback function for each event type to detect. When the application starts a communication, it can submit the requests to poll (1) and wait for them or simply continue its computation. Periodically, the thread scheduler calls the I/O manager (2) in order to poll the network by calling the callback functions (3).

We propose to manage the communication events in a dedicated controller linked to the thread scheduler for several reasons. Firstly, centralizing avoids the concurrency issues encountered when several threads try to poll the same network. Since the I/O manager has a global view of the pending requests, it can poll each request one after another. Moreover, the manager has the opportunity to aggregate multiple requests. If several threads are waiting for messages on a single network interface, it can be interesting to aggregate these requests when polling.

Secondly, the thread scheduler has the opportunity to preempt a computing task and call the I/O manager in order to detect a potential network I/O completion and thus make the communication progress. This is useful when the

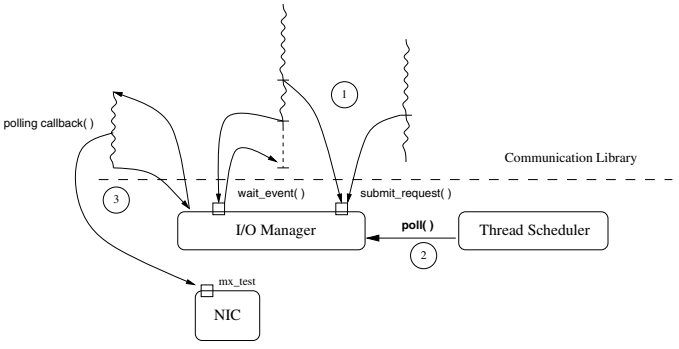


Fig. 1. Example of interaction between the I/O manager and the MPI library

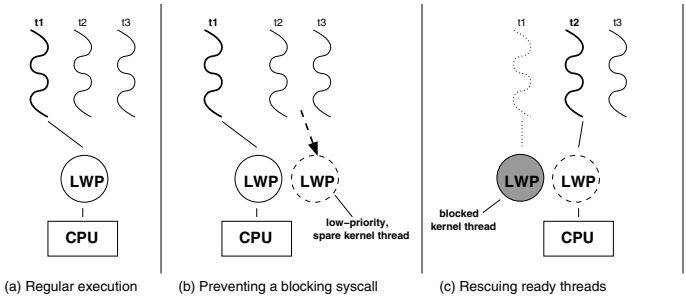


Fig. 2. Low priority, spare kernel-level threads are used to schedule remaining application threads in case a blocking syscall occurs during an I/O critical operation

application performs asynchronous operations that require some processing once the communication ends. For example, in a *rendez-vous* protocol, the receiver has to post a receiving request to synchronize with the sender. Once both sides are synchronized, the transfer can start: one side receives the data that the other side sends. In that case, the progression offered by the I/O manager and the thread scheduler allows to completely overlap the communication with computation.

3.2 Passive Waiting: Interrupts

Passive monitoring through blocking system calls is tricky to implement in a two-level scheduler. Indeed, during regular execution of application threads, our scheduler binds exactly one kernel thread (also called LightWeight process – LWP) per processor (Fig. 2-a), so that the scheduling of threads can be entirely performed at the user-level. A blocking system call could therefore prevent a whole subset of user-level threads to run. To avoid this and keep reactivity low, we proceed as follows.

Before executing a (potentially blocking) I/O system call, the client thread first wakes up *a spare kernel thread* (Fig. 2-b) to shepherd the remaining ready threads on the underlying processor. Since this kernel thread runs at a very low

priority, it will not be scheduled until the previous kernel thread blocks. Thus, *if the system call completes without blocking*, the I/O client will continue its execution with a very high priority, as requested. At the end of the I/O section, the spare kernel thread simply returns to the "sleep" state. On the opposite, *if the call blocks*, the original kernel thread yields the CPU to the spare one (Fig. 2-c). Upon I/O completion, the NIC interrupt handler will wake up the original kernel thread that will, in turn, immediately continue the execution of the client thread. This way, the reactivity of the client thread is optimal.

Note that no modification to the underlying operating system is required, as opposed to solutions such as *Scheduler Activations* [9,11].

3.3 Active Waiting: Polling

In implementing active polling, our system carefully cooperates with the thread scheduler to avoid busy waiting and unnecessary context switches. Applications register new types of I/O events with some polling trigger(s) (at every context switch, after a period of time, when a CPU gets idle, etc.) The thread scheduler then invokes the I/O manager accordingly. However, these invocations occur in a restricted context with some classes of actions being prohibited (synchronization primitives, typically). Thus, they are similar to interrupt handlers within an operating system.

Most of the I/O manager code is consequently run *outside* the restricted context in the form of *tasklets* [12]. Tasklets have been introduced in operating systems to defer treatments that cannot be performed within an interrupt handler. They run as soon as possible (they have a very high priority) when the scheduler reaches a point where it is safe to run tasklets. They have additional properties. Firstly, tasklets of the same type run under mutual exclusion, which simplifies the I/O manager code and even makes it more efficient. Secondly, the execution of tasklets can be enforced on a particular processor, which allows to maximize cache affinity by running tasklets on the same processor as their client thread.

3.4 Handling of Both Interrupts and Polling

Most of the network interfaces (MX/Myrinet, Infiniband Verbs, TCP sockets) provide both polling and interrupt-based functions to detect network I/O events. To ensure a good reactivity, our I/O manager uses one method or the other depending on the context: number of running threads and available CPUs. This kind of strategy has already been developed in Panda [13], but ours also takes into account the upper layer's preference: the communication library or the application has full knowledge of the request completion time. A smarter approach could also take into account the history of requests or their priorities. A similar method was developed in *polling watchdog*[14] but it required a specific kernel support.

Table 1. Benchmark program for MPI asynchronous progression

Sender	*Receiver*

`get_time(t1);`
`MPI_Send(...);`
`get_time(t2);`
	`MPI_IRecv(...);`
	`compute();`
	`/* approx. 50ms. computation */`
	`MPI_Wait(...);`

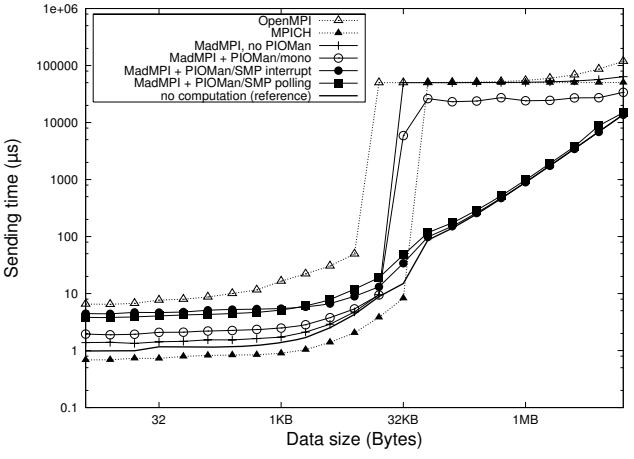


Fig. 3. `MPI_Send` time with MX

4 Evaluation

We have evaluated the implementation of our I/O manager using the NewMadeleine [1] communication library and its built-in MPI implementation called Mad-MPI. The point-to-point nonblocking posting (`isend`, `irecv`) and completion (`wait`, `test`) operations of Mad-MPI are directly mapped to the equivalent operations of NewMadeleine. We performed benchmarks that evaluate the MPI asynchronous operation progression in background (communication/computation overlap) and benchmarks that evaluate the overhead of PIOMan. All these experiments have been carried out on a set of two dual-core 1.8 GHz Opteron boxes interconnected through Myri-10G NICs with the MX1.2.1 driver providing a latency of 2.3μs.

MPI asynchronous progression of communications. To evaluate the MPI asynchronous progression, we use the benchmark program listed on Table 1. This program attempts to overlap communication and computation on the receiver side. We record the time spent in sending and we compare the results to a reference obtained with Mad-MPI.

Figure 3 shows the sending time (time spent in `MPI_Send`) we measured over MX/Myrinet with Mad-MPI, OpenMPI 1.2.1, and MPICH/MX 1.2.7. We measured similar results over other network types (Infiniband and TCP). For small

Table 2. PIOMan's average overhead

	no thread	mono	SMP
polling	0.038 μs	0.085 μs	0.142 μs
interrupt	-	-	1.68 μs

messages, all implementations show a sending time close to the network latency. For larger messages, when a *rendez-vous* is performed, we observe three different behaviors:

no asynchronous progress – OpenMPI and plain Mad-MPI do not support background progress of *rendez-vous* handshake. Therefore, the sender is blocked until the receiver reaches the `MPI_Wait`. MPICH makes the handshake progress thanks to the MX progression thread but in the current implementation, the notification of the transfer is not overlapped.

coarse grained interleaved progress – PIOMan/mono tasklets are scheduled upon timer interrupt, every 10 ms. We observe that the delay to complete the *rendez-vous* is now bounded by 10 ms instead of the full computation time.

full overlap – PIOMan/SMP is able to schedule tasklets on another LWP, thus we get a full overlap of communication and computation. We observe on the figure that the *rendez-vous* performance does not suffer from the computation on the receiver side.

We conclude that PIOMan is able to actually overlap MPI communication and computation while OpenMPI, MPICH, and plain Mad-MPI were not able to make communication progress asynchronously.

Overhead evaluation. We have evaluated the overhead of the I/O manager with empty polling and blocking functions. The results are shown in Table 2. The polling overhead differs from one version to the other. This is due to the cost of synchronization being different over each version. The interrupt overhead has only been evaluated on the SMP version since only this version implements the mechanism. We observe that the overhead is negligible for polling. On the other hand, the cost of blocking calls (interrupts) is quite high due to the awakening of the sleeping LWP and the communication between LWPs. However, interrupts are supposed to be used when the CPU is doing computation, where the delay would have been several order of magnitude higher without interrupts.

5 Conclusions and Future Work

Overlapping MPI communications and computation do matter if we do not want to waste thousands of CPU cycles. However, making communications progress efficiently is not so simple as adding a communication thread. In this paper, we have proposed a generic and portable communication events manager that is able to actually overlap communication and computation. This I/O manager is able to

handle both active polling and interrupts and integrates gracefully with our multithreading scheduler. We obtained effective communication/computation overlapping with our I/O manager, as opposed to other widespread MPIs.

In the near future, we plan to use PIOMan inside other MPI implementations such as MPICH-2 or communication frameworks like PadicoTM. We also intend to make a more efficient use of NUMA architectures by trying to execute polling tasklets on the most suitable CPU given the architecture topology.

References

1. Aumage, O., Brunet, E., Furmento, N., Namyst, R.: Newmadeleine: a fast communication scheduling engine for high performance networks. In: CAC 2007: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007 (2007)
2. Sancho, J.C., et al.: Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In: SC 2006, IEEE Computer Society, Los Alamitos (2006)
3. Doerfler, D., Brightwell, R.: Measuring MPI send and receive overhead and application availability in high performance network interfaces. In: Euro PVM/MPI, pp. 331–338 (2006)
4. ANL, MCS Division (2007), MPICH-2 Home Page http://www.mcs.anl.gov/mpi/mpich/
5. The Open MPI Project: Open MPI: Open Source High Performance Computing (2007), http://www.open-mpi.org/
6. Graham, R.L., et al.: Open MPI: A high-performance, heterogeneous MPI. In: Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Barcelona, Spain (2006)
7. Quadrics Ltd.: Elan Programming Manual (2003), http://www.quadrics.com/
8. Myricom Inc.: Myrinet EXpress (MX): A High Performance, Low-level, Message-Passing Interface for Myrinet (2003), http://www.myri.com/scs/.
9. Anderson, T.E., Bershad, B.N., Lazowska, E.D., Levy, H.M.: Scheduler activations: effective kernel support for the user-level management of parallelism. ACM Trans. Comput. Syst. 10(1), 53–79 (1992)
10. Runtime Team, LaBRI-Inria Futurs: Marcel: A POSIX-compliant thread library for hierarchical multiprocessor machines (2007), http://runtime.futurs.inria.fr/marcel/
11. Danjean, V., Namyst, R., Russell, R.: Integrating kernel activations in a multithreaded runtime system on Linux. In: Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00) (2000)
12. Russel, P.: Unreliable guide to hacking the linux kernel (2000)
13. Langendoen, K., Romein, J., Bhoedjang, R., Bal, H.: Integrating polling, interrupts, and thread management. frontiers 00, 13 (1996)
14. Maquelin, O., et al.: Polling watchdog: combining polling and interrupts for efficient message handling. In: ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture (1996)

Investigations on InfiniBand: Efficient Network Buffer Utilization at Scale

Galen M. Shipman[1], Ron Brightwell[2], Brian Barrett[1], Jeffrey M. Squyres[3], and Gil Bloch[4]

[1] Los Alamos National Laboratory*, Los Alamos, NM USA, LA-UR-07-3198
{gshipman,bbarrett}@lanl.gov
[2] Sandia National Laboratories**, Albuquerque, NM USA
rbbrigh@sandia.gov
[3] Cisco, Inc., San Jose, CA USA
jsquyres@cisco.com
[4] Mellanox Technologies, Santa Clara, CA USA
gil@mellanox.com

Abstract. The default messaging model for the OpenFabrics "Verbs" API is to consume receive buffers in order—regardless of the actual incoming message size—leading to inefficient registered memory usage. For example, many small messages can consume large amounts of registered memory. This paper introduces a new transport protocol in Open MPI implemented using the existing OpenFabrics Verbs API that exhibits efficient registered memory utilization. Several real-world applications were run at scale with the new protocol; results show that global network resource utilization efficiency increases, allowing increased scalability—and larger problem sizes—on clusters which can increase application performance in some cases.

1 Introduction

The recent emergence of near-commodity clusters with thousands of nodes connected with InfiniBand (IB) has increased the need for examining scalability issues with MPI implementations for IB. Several of these issues were originally discussed in detail for the predecessor to IB [1], and several possible approaches to overcoming some of the more obvious scalability limitations were proposed. This study examines the scalability, performance, and complexity issues of the message buffering for implementations of MPI over IB.

The semantics of IB Verbs place a number of constraints on receive buffers. Receive buffers are consumed in FIFO order, and the buffer at the head of

* Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396.

** Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 178–186, 2007.

the queue must be large enough to hold the next incoming message. If there is no receive buffer at the head of the queue or if the receive buffer is not large enough, this will trigger a network-level protocol that can significantly degrade communication performance. Because of these constraints, MPI implementations must be careful to insure that a sufficient number of receive buffers of sufficient size are always available to match incoming messages.

Several other details can complicate the issue. Most operating systems require message buffers to be registered so that they may be mapped to physical pages. Since this operation is time consuming, it is desirable to register memory only when absolutely needed and to use registered memory as efficiently as possible. MPI implementations that are single-threaded may only be able to replenish message buffers when the application makes an MPI library call. Finally, the use of receive buffers is highly dependent on application message passing patterns.

This paper describes a new protocol that is more efficient at using receive buffers and can potentially avoid some of the flow control protocols that are needed to insure that receive buffers of the appropriate size are always available. We begin by measuring the efficiency of receive buffer utilization for the current Open MPI implementation for IB. We then propose a new strategy that can significantly increase the efficiency of receive buffer utilization, make more efficient use of registered memory, and potentially reduce the need for MPI-level flow control messages.

The rest of this paper is organized as follows. The next section provides a brief discussion of previous work in this area. Section 3 describes the new protocol, while Section 4 provides details of the test platform and analyzes results for several application benchmarks and applications. We conclude in Section 5 with a summary of relevant results and offer some possible avenues of future work in Section 5.1.

2 Background

The complexity of managing multiple sets of buffers across multiple connections was discussed in [1], and a mechanism for sharing a single pool of buffers across multiple connections was proposed. In the absence of such a shared buffer pool for IB, MPI implementations were left to develop user-level flow control strategies to insure that message buffer space was not exhausted [2].

Eventually, a shared buffer pool mechanism was implemented for IB in the form of a shared receive queue (SRQ), and has been shown to be effective in reducing memory usage [3, 4]. However, the IB SRQ still does not eliminate the need for user-level and network-level flow control required to insure the shared buffer pool is not depleted [5]. The shared buffer pool approach was also implemented for MPI on the Portals data movement layer, but using fixed sized buffers was shown to have poor memory efficiency in practice, so an alternative strategy was developed [6]. In Section 3, we propose a similar strategy that can be used to improve memory efficiency for MPI over IB as well.

3 Protocol Description

IB does not natively support receive buffer pools similar to [6], but it is possible to emulate the behavior with buckets of receive buffers of different sizes, with each bucket using a single shared receive queue (SRQ). We call this emulation the "Bucket SRQ," or B-SRQ.

B-SRQ begins by allocating a set of per-peer receive buffers. These per-peer buffers are for "tiny" messages (128 bytes plus space for header information) and are regulated by a simple flow control protocol to ensure that tiny messages always have a receive buffer available.[1] The "tiny" size of 128 bytes was choosen as an optimization to ensure that global operations on a single `MPI_DOUBLE` element would always fall into the per-peer buffer path. The 128-byte packet size also ensures that other control messages (such as rendezvous protocol acknowledgments) use the per-peer allocated resources path as well.

B-SRQ then allocates a large "slab" of registered memory for receive buffers. The slab is divided up into N buckets; bucket B_i is S_i bytes long and contains a set of individual receive buffers, each of size R_i bytes (where $R_i \neq R_j$ for $i, j \in [0, N-1]$ and $i \neq j$). Bucket B_n contains $\frac{S_n}{R_n}$ buffers. Each bucket is associated with a unique queue pair (QP) and a unique SRQ. This design point is a result of current IB network adapter limitations; only a single receive queue can be associated with a QP. The QP effectively acts as the sender-side addressing mechanism of the corresponding SRQ bucket on the receiver.[2] Other networks such as Myrinet GM [7] allow the sender to specify a particular receive buffer on the send side through a single logical connection and as such would allow for a similar protocol as that used in B-SRQ. Quadrics Elan [8] uses multiple pools of buffers to handle unexpected messages, although the specific details of this protocol have never been published.

In our prototype implementation, $S_i = S_j$ for $i \neq j$, and $R_i = 2^{8+i}$, for $i \in [0, 7]$. That is, the slab was divided equally between eight buckets, and individual receive buffers were powers of two sizes ranging from 2^8 to 2^{15}. Fig. 1 illustrates the receive buffer allocation strategy.

Send buffers are allocated using a similar technique, except that free lists are used which grow on demand (up to a configurable limit). When a MPI message is scheduled for delivery, a send buffer is dequeued from the free list; the smallest available send buffer that is large enough to hold the message is returned.

4 Results

Two protocols are analyzed and compared: Open MPI v1.2's default protocol for short messages and Open MPI v1.2's default protocol modified to use B-SRQ for short messages (denoted "v1.2bsrq").

[1] The flow control protocol, while outside the scope of this paper, ensures that the receiver never issues a "receiver not ready" (or RNR-NAK) error, which can degrade application performance.

[2] Open MPI uses different protocols for long messages. Therefore, the maximum B-SRQ bucket size is effectively the largest "short" message that Open MPI will handle.

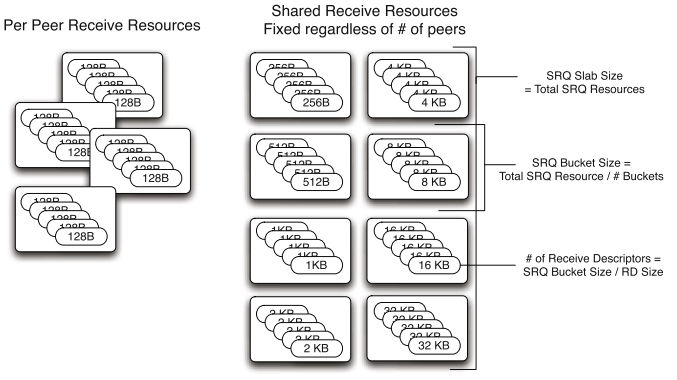


Fig. 1. Open MPI's B-SRQ receive resources

The default protocol for short messages in Open MPI v1.2 uses a credit-based flow-control algorithm to send messages to fixed-sized buffers on the receiver. When the sender exhausts its credits, messages are queued until the receiver returns credits (via an ACK message) to the sender. The sender is then free to resume sending. By default, SRQ is not used in Open MPI v1.2 because of a slight latency performance penalty; SRQ may be more efficient in terms of resource utilization, but it can be slightly slower than normal receive queues in some network adapters [3, 4].

Open MPI v1.2bsrq implements the protocol described in Section 3. It has the following advantages over Open MPI v1.2's default protocol:

- Receiver buffer utilization is more efficient.
- More receive buffers can be be posted in the same amount of memory.
- No flow control protocol overhead for messages using the SRQ QPs.[3]

4.1 Experimental Setup

The Coyote cluster at Los Alamos National Laboratory was used to test the B-SRQ protocol. Coyote is a 1,290 node AMD Opteron cluster divided into 258-node sub-clusters. Each sub-cluster is an "island" of IB connectivity; nodes are fully connected via IB within the sub-cluster but are physically disjoint from IB in other sub-clusters. Each node has two single-core 2.6 GHz AMD Opteron processors, 8 GB of RAM, and a single-port Mellanox Sinai/Infinihost III SDR IB adapter (firmware revision 1.0.800). The largest MPI job that can be run utilizing the IB network is therefore 516 processors.

Both versions of Open MPI (v1.2 and v1.2bsrq) were used to run the NAS Parallel Benchmarks (NPBs) and two Los Alamos-developed applications. Wall-clock execution time was measured to compare overall application performance with and without B-SRQ. Instrumentation was added to both Open MPI versions

[3] Without pre-negotiating a fixed number of SRQ receive buffers for each peer, there is no way for senders to know when receive buffers have been exhausted in an SRQ [3].

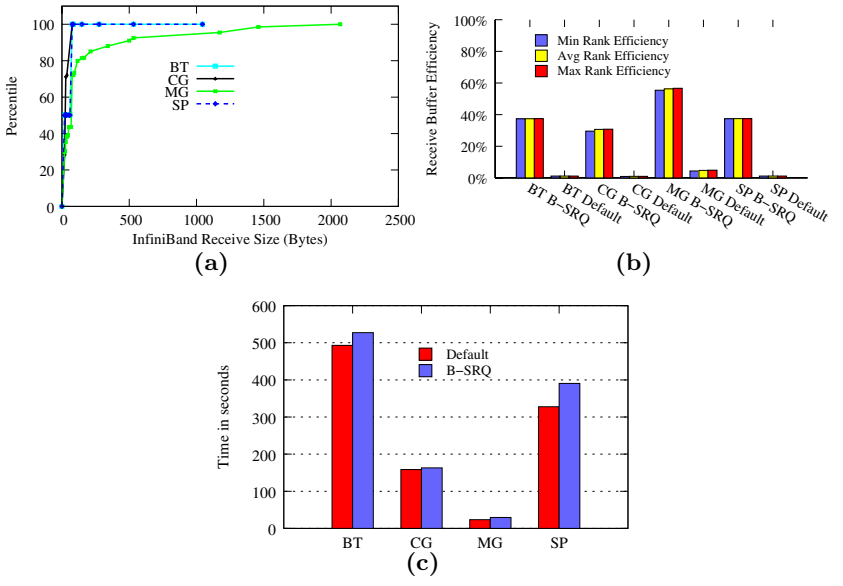


Fig. 2. NAS Parallel Benchmark Class D results on 256 nodes for (a) message size, (b) buffer efficiency, and (c) wall-clock execution time

to measure the effectiveness and efficiency of B-SRQ by logging send and receive buffer utilization, defined as $\frac{message\_size}{buffer\_size}$. Performance results used the non-instrumented versions.

Buffer utilization of 100% is ideal, meaning that the buffer is exactly the same size as the data that it contains. Since IB is a resource-constrained network, the buffer utilization of an application can have a direct impact on its overall performance. For example, if receive buffer utilization is poor and the incoming message rate is high, available receive buffers can be exhausted, resulting in an RNR-NAK (and therefore performance degradation) [3, 4].

The frequency of message sizes received via IB was also recorded. Note that IB-level message sizes may be different than MPI-level message sizes as Open MPI may segment an MPI-level message, use RDMA for messages, and send control messages over IB. This data is presented in percentile graphs, showing the percentage of receives at or below a given size (in bytes).

4.2 NAS Parallel Benchmarks

Results using `D` sized problems with the NPBs are shown in Figure 2(a). The benchmarks utilize a high percentage of small messages at the IB level, with the notable exception of MG, which uses some medium-sized messages at the IB level. Some of these benchmarks do send larger messages as well, triggering a different message passing protocol in Open MPI that utilizes both rendezvous techniques and RDMA [9]. Long messages effectively avoid the use of dedicated receive buffers delivering data directly into the application's receive buffer.

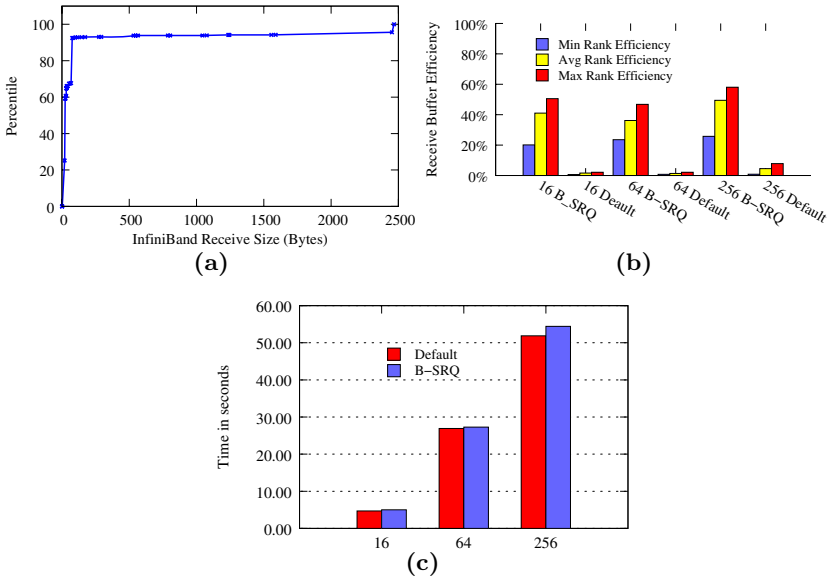


Fig. 3. SAGE results for (a) message size at 256 processes, (b) buffer efficiency at varying process count, and (c) wall-clock execution time at varying process count

Figure 2(b) shows that Open MPI's v1.2 protocol exhibits poor receive buffer utilization due to the fact that receive buffers are fixed at 4 KB and 32 KB. These buffer sizes provide good performance but poor buffer utilization. The B-SRQ protocol in v1.2bsrq provides good receive buffer utilization for the NPB benchmarks, with increases in efficiency of over 50% for the MG benchmark. Overall B-SRQ performance decreases slightly, shown in Figure 2(c). Measurements of class B and C benchmarks at 256 processes do not exhibit this performance degradation which is currently being investigated in conjunction with several IB vendors.

4.3 Los Alamos Applications

Two Los Alamos National Laboratory applications were used to evaluate B-SRQ. The first, SAGE, is an adaptive grid Eulerian hydrocode that uses adaptive mesh refinement. SAGE is typically run on thousands of processors and has weak scaling characteristics. Message sizes vary, typically in the tens to hundreds of kilobytes. Figure 3(a) shows that most IB level messages received were less than 128 bytes with larger messages using our RDMA protocol. Figure 3(b) illustrates poor memory utilization in Open MPI v1.2 when run at 256 processors. The new protocol exhibits much better receive buffer utilization, as smaller buffers are used to handle the data received. As with the NPBs, there is a slight performance impact, but it is very minimal at 256 processors, as illustrated in Figure 3(c).

The second application evaluated was SWEEP. SWEEP uses a pipelined wavefront communication pattern. Messages are small and communication in the wavefront is limited to spatial neighbors on a cube. The wavefront direction and

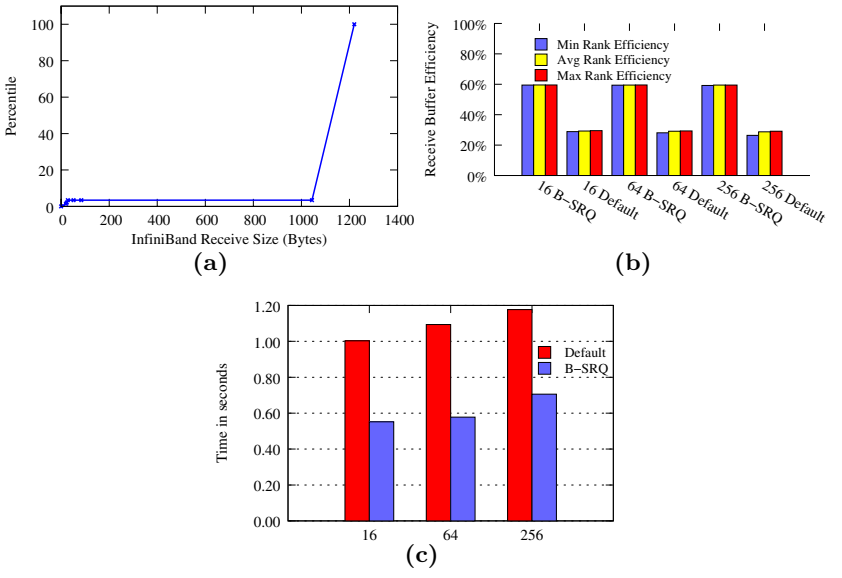


Fig. 4. SWEEP results for (a) message size at 256 processes, (b) buffer efficiency at varying process count, and (c) wall-clock execution time at varying process count.

starting point is not fixed; it can change during the run to start from any corner on the cube. Receive buffer size is highly regular as shown in Figure 4(a). Figure 4(b) shows that receive buffer efficiency is improved by the B-SRQ algorithm, although to a lesser extent than in other applications. Figure 4(c) shows that performance was dramatically increased by the use of the B-SRQ protocol. This performance increase is due to the B-SRQ protocol providing a much larger number of receive resources to any one peer (receive resources are shared). The v1.2 protocol provides a much smaller number of receive resources to any one peer (defaulting to 8). This small number of receive resources prevents SWEEP's high message rate from keeping the IB network full. The B-SRQ protocol allows SWEEP to have many more outstanding send/receives as a larger number of receive resources are available to a rank's 3-D neighbors and thereby increasing performance. This performance increase could also be realized using v1.2's per peer allocation policy but would require allocation of a larger number of per peer resources to each process. B-SRQprovides good performance without such application specific tuning while conserving receive resources.

5 Conclusions

Evaluation of application communication over Open MPI revealed that the size of data received at the IB level, while often quite small, can vary substantially from application to application. Using a single pool of fixed-sized receive buffers may therefore result in inefficient receive buffer utilization. Receive buffer depletion

negatively impacts performance, particularly at large scale. Earlier work focused on complex receiver-side depletion detection and recovery, or avoiding depletion altogether by tuning receive buffer sizes and counts on a per-application (and sometimes per-run) basis.

The B-SRQ protocol focuses on avoiding resource depletion by more efficient receive buffer utilization. Experimental results are promising; by better matching receive buffer sizes with the size of received data, application runs were constrained to an overall receive buffer memory footprint of less than 25 MB. Additionally, receive queues were never depleted and no application-specific tuning of receive buffer resources was required.

5.1 Future Work

While the B-SRQ protocol results in better receive buffer utilization, other receive buffer allocation methods are possible. For example, buffer sizes in increasing powers of two may not be optimal. Through measurements collected via Open MPI v1.2bsrq, we will explore receive buffer allocation policies and their potential impact on overall receive buffer utilization.

Recent announcements by IB vendors indicate that network adapters will soon support the ability to associate a single QP with multiple SRQs by specifying the SRQ for delivery on the send side. The ConnectX IB HCA (an InfiniBand Host Channel Adapter recently announced by Mellanox) extends the InfiniBand specification and defines a new Transport Service called SRC (Scalable Reliable Connection). SRC transport service separates the transport context from the Receive Work Queue and allows association of multiple receive-queues to a single connection. Using SRQ Transport Service, the sender indicates the destination receive queue when posting a send request to allow de-multiplexing in the remote receiver. Using the new paradigm, an efficient BSRQ can be implemented using a single connection context and a single send queue to send data to different remote SRQs. When posting a work request the sender will indicate the appropriate destination SRQ according to message size (and priority).

References

[1] Brightwell, R., Maccabe, A.B.: Scalability limitations of VIA-based technologies in supporting MPI. In: Proceedings of the Fourth MPI Devlopers' and Users' Conference (2000)

[2] Liu, J., Panda, D.K.: Implementing efficient and scalable flow control schemes in mpi over infiniband. In: Workshop on Communication Architecture for Clusters (CAC 04) (2004)

[3] Shipman, G.M., Woodall, T.S., Graham, R.L., Maccabe, A.B., Bridges, P.G.: Infiniband scalability in Open MPI. In: International Parallel and Distributed Processing Symposium (IPDPS'06) (2006)

[4] Sur, S., Chai, L., Jin, H.W., Panda, D.K.: Shared receive queue based scalable MPI design for InfiniBand clusters. In: International Parallel and Distributed Processing Symposium (IPDPS'06) (2006)

[5] Sur, S., Koop, M.J., Panda, D.K.: High-performance and scalable MPI over InfiniBand with reduced memory usage: An in-depth performance analysis. In: ACM/IEEE International Conference on High-Performance Computing, Networking, Storage, and Analysis (SC'06) (2006)
[6] Brightwell, R., Maccabe, A.B., Riesen, R.: Design, implementation, and performance of MPI on Portals 3.0. International Journal of High Performance Computing Applications 17(1) (2003)
[7] Myrinet: Myrinet GM http://www.myri.com/scs/documentation.html
[8] Quadrics: Quadrics elan programming manual v5.2 (2005), http://www.quadrics.com/
[9] Shipman, G.M., Woodall, T.S., Bosilca, G., Graham, R.L., Maccabe, A.B.: High performance RDMA protocols in HPC. In: Proceedings, 13th European PVM/MPI Users' Group Meeting, Bonn, Germany, Springer, Heidelberg (2006)

Improving MPI Support for Applications on Hierarchically Distributed Resources

Raúl López and Christian Pérez

IRISA/INRIA, Campus de Beaulieu, 35042 Rennes cedex, France
{Raul.Lopez,Christian.Perez}@inria.fr

Abstract. Programming non-embarrassingly parallel scientific computing applications such as those involving the numerical resolution of system of PDEs using mesh based methods for grid computing environments is a complex and important issue. This work contributes to this goal by proposing some MPI extensions to let programmers deal with the hierarchical nature of the grid infrastructure thanks to a tree representation of the processes as well as the corresponding extension of collective and point-to-point operations. It leads in particular to support $N \times M$ communications with transparent data redistribution.

Keywords: MPI, Grids, tree structure, hierarchical communication, data redistribution.

1 Introduction and Motivation

Grid computing is currently the subject of a lot of research activities worldwide. It is particularly well suited to compute intensive, embarrassingly parallel scientific computing applications. The situation is less clear for non-embarrassingly parallel scientific computing applications such as those involving the numerical resolution of systems of PDEs (partial differential equations) using mesh based (finite difference, finite volume or finite element) methods. In most cases, grid-enabled numerical simulation tools are essentially a direct porting of parallel software previously developed for homogeneous machines, thanks to the availability of appropriate MPI implementations such as MPICH-G2[1]. However, these grid-enabled simulation software rarely take into account important architectural issues characterizing computational grids, such as heterogeneity both of processing nodes and interconnection networks, which have a great impact on performance. Moreover, they are quite difficult to program. Considering that a computational grid can be seen as a hierarchical architecture, the objective of the present work is to improve the support of MPI based scientific computing applications on hierarchically distributed resources. An extended API is proposed to deal with the hierarchical structure of resources with respect to the dynamic discovery of the hierarchical resource properties and to the generalization of point-to-point and collective communications to this hierarchical structure.

This work is taking place in the context of the DISC project[1] which targets to demonstrate that a computational grid can be used as a high performance computing platform

[1] Project number ANR-05-CIGC-005 funded by the French ANR under the framework of the 2005 program *Calcul Intensif et Grilles de Calcul.*

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 187–194, 2007.

for non-embarrassingly parallel scientific computing applications. Though DISC focuses on component models, this paper aims at showing that advanced concepts may also be applied to MPI.

The remainder of this paper is divided as follows. Section 2 introduces a class of motivating application and Section 3 presents the related works. The proposed MPI extensions are dealt with in Section 4 while Section 5 concludes the paper.

2 Motivating Application and Infrastructure

2.1 Motivating Application

The targeted scientific computing applications take the form of three-dimensional finite element software for the simulation of electromagnetic wave propagation (computational electromagnetism - CEM) and compressible fluid flow (computational fluid dynamics - CFD). The traditional way of designing scalable parallel software for the numerical resolution of systems of PDEs is to adopt a SPMD programming model that combines an appropriate partitioning of the computational mesh (or the algebraic systems of equations resulting from the time and space discretization of the systems of PDEs defining the problem) and a message passing programming model while portability is maximized through the use of the Message Passing Interface (MPI). It is clear that such an approach does not directly transpose to computational grids despite the fact that appropriate, interoperable, implementations of MPI have been developed (e.g. MPICH-G2[1] in the Globus Toolkit[2]). By this, we mean that computational grids raise a number of issues that are rarely faced with when porting a scientific computing application relying on programming models and numerical algorithms devised for classical parallel systems. The most important of these issues with regards to parallel performances is heterogeneity (both of processing nodes and interconnection networks which impacts computation and communication performances). This heterogeneity issue is particularly challenging for unstructured mesh based CEM and CFD solvers because of two major issues. First, they involve iterative solution methods and time stepping schemes that are mostly synchronous thus requiring high performance networks to achieve scalability for a large number of processing nodes. Second, most if not all of the algorithms used for the partitioning of computational meshes (or algebraic systems of equations) do not take into account the variation of the characteristics of processing nodes and interconnection networks.

2.2 Infrastructure

A grid computing platform such as the Grid'5000 experimental test-bed can be viewed as a (at least) three level architecture: the highest level consists of a small number ($<$ 10) of clusters with between a few hundreds and some thousand nodes. These clusters are interconnected by a wide area network (WAN) which, in the case of Grid'5000 takes the form of dedicated 10 Gb/s links provided by Renater. The intermediate level is materialized by the system area network (SAN) connecting the nodes of a given cluster. This level is characterized by the fact that the SAN may differ from one cluster to the other (Gigabit Ethernet, Myrinet, Infiniband, etc.). The lowest level is used to

represent the architecture of a node, single versus multiple-processor systems, single versus multiple-core systems and memory structure (SMP, Hypertransport, etc.).

2.3 Discussion

In the context of mesh based numerical methods for systems of PDEs, one possible alternative to the standard parallelization strategy consists in adopting a multi-level partitioning of the computational mesh. Hence, we can assume that the higher level partitioning is guided by the total number of clusters while the lower level partitioning is performed locally (and concurrently) on each cluster based on the local numbers of processing nodes. Considering the case of two-level partitioning, two types of artificial interfaces are defined: intra-level interfaces refer to the intersection between neighboring subdomains which belong to the same first-level partition while inter-level interfaces denote interfaces between two neighboring subdomains which belong to different first-level subdomains. Then, the parallelization can proceed along two strategies.

A first strategy is to use a flat (or single-level) parallelization essentially exploiting the lower level partitioning and where the intra- and inter-level interfaces are treated in the same way. A second strategy consists in using a hierarchical approach where a distinction is made in the treatment of intra- and inter-level interfaces. In this case, we can proceed in two steps: (1) the exchange of information takes place between neighboring subdomains and in the higher level partitioning through inter-level interfaces, which involves a $N \times M$ redistribution operation; (2) within each higher level subdomain, the exchange of information takes place between neighboring subdomains in the lower level partitioning through the intra-level interfaces.

3 Related Works

As far as we know, the related works can be divided in two categories. First, several efforts such as MPICH-G2 [1], OpenMPI [3], or GridMPI [4], have been done to provide efficient implementations of the flat MPI model on hierarchical infrastructures. The idea is to make use of hierarchy-aware algorithms and of a careful utilization of the networks for implementing collective communications.

In these works, the MPI runtime is aware of the hierarchy of resources but not the programmers. As this is a limiting factor for hierarchical applications like those presented in Section 2, there has been a proposal to extend the MPI model: MPICH-G2 provides an extended MPI with two attributes associated with every communicator to let the application discover the actual topology. The depths define the available levels of communication for each process, noting that MPICH-G2 defines 4 levels (WAN-TCP, LAN-TCP, intra-machine TCP, and native MPI). The colors are a mechanism that determines whether two processes can communicate at a given level. While such a mechanism helps the programmers in managing the resource topology, it suffers several limitations: there is a hard coded number of levels so that some situations cannot be taken into account, like NUMA nodes[2]. Moreover, a communication infrastructure is associated with each level while application developers care more about network performance parameters like latency or bandwidth.

[2] Communication performance within a NUMA node is not the same as through a network.

A second set of works deals with $N \times M$ communications such as InterComm [5] or PaCO++ [6]. Their goal is to relieve programmers from complex, error prone and resource dependent operations that involve data redistribution among processes. These works study how to easily express data distributions for programmers so as to automatically compute an efficient communication schedule with respect to the underlying network properties. They appear to be a complement to MPI, as they take into account communications from parallel subroutine to parallel subroutine, that is to say that they can be observed as the parallel extension of MPI point to point communications. Note that there are not yet standard APIs to describe data distributions, though there have been some standardization efforts to provide a common interface like DRI [7].

4 Improving MPI Support for Hierarchical Resources

Relying on hierarchical resources has become a major goal in the message passing paradigm as it has been exposed. However, the support given to users by existing frameworks does not always match their needs, as long as program control, data-flow and infrastructure's attributes are independently handled. This section presents an extension to the MPI API to improve such a situation.

4.1 Point-to-Point and Collective Communications

MPI modifications proposed here intend to bring together the advantages from the message passing paradigm and the hierarchical view of a computing system by the programmer. Those modifications are based on the utilization of a simple tree model, where processes are the leafs of the tree while nodes are just an abstraction of groups of processes (organized according to their topology). More precisely this tree can be observed as the hierarchical organization of resources at run-time. Figure 1 shows an example of such structure together with the hierarchically partitioned data used by an application.

Hierarchical identification system. We propose to introduce an alternate rank definition, called `hrank` to denote its hierarchical nature. It consists of an array of identifiers that represents the tree nodes in the different levels to which the process belongs from the root to the bottom of the tree. This representation intrinsically involves the tree organization and easily identifies the resource grouping. In Figure 1, each node of the tree has been associated with its hrank. For example, the process of MPI rank 6 has `0.1.1` as hrank.

Navigation of the tree structure. The programmer is given a set of functions to explore the tree structure and the characteristics of nodes and leafs. This allows him/her to take fine grain decisions regarding which resource carries out each computing task or how data are partitioned. The tree structure can be browsed thanks to functions like `HMPI_GET_PARENT(hrank)`, `HMPI_GET_SONS(hrank)`, `HMPI_GET_SIBLINGS(hrank)`, or `HMPI_GET_LEVEL(hrank)`. Such functions return either an hrank or an array of hranks.

Moreover, the programmer can also get access to some dynamic properties of the physical links between nodes and leafs. For example, the extended API may provide the

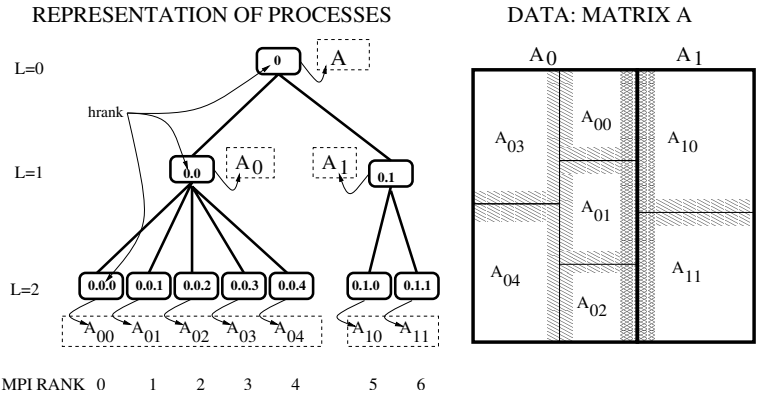


Fig. 1. *Left:* Representation of processes derived from the structure of resources, last row indicates the rank in a standard MPI communicator. *Right:* Data hierarchical partitioning. Dashed rectangles on the left show the association of data to resources.

functions `HMPI_GET_LATENCY(hrank1, hrank2)` or `HMPI_GET_BANDWITH (hrank1, hrank2)` to obtained the current values.

Collective communications in the hierarchical model. The collective operations are slightly modified to take advantage of the existing hierarchy. Concretely a new parameter is added to indicate the group of processes involved in the operation. Such a parameter is the hrank of the node defining the subtree where to execute the collective operation. With respect to Figure 1, the call `HMPI_REDUCE (sendbuf, recvbuf, count, datatype, operation, root, HMPI_COMM_WORLD, 0.1)` performs a reduction involving all processes belonging to the subtree whose root is the node `0.1`, that is to say the reduction is restricted to the two last processes (`0.1.0` and `0.1.1`).

Point to point communications in the hierarchical model. Elementary point to point operations – those involving a single sender and a single receiver – keep their semantics and only the way a process is identified changes by the usage of hranks instead of ranks. These operations are called `HMPI_SEND`, `HMPI_RECV`, etc. In Figure 1, a former `MPI_SEND` from process ranked 1 to process ranked 5 is equivalent to `HMPI_SEND` to `0.1.0` called by the process with hrank `0.0.1`.

Nonetheless the most interesting feature is that this hierarchical representation enables using MPI communications at a higher level. For instance, suppose that in Figure 1, `0.0` sends some data to `0.1`. In other words, the processes `0.0.`x ($x \in \{0, 4\}$) call `HMPI_SEND` with `0.1` as destination. Symmetrically, processes `0.1.`x ($x \in \{0, 1\}$) invoke `HMPI_RECV` with `0.0` as source. It involves a $N \times M$ (here a 5×2) communication and data distribution issues become transparent for programmers. The required data distribution support is discussed in the next section.

Communications between any two levels are also possible. A tree node in level 1, as for instance `0.1`, can send data to a tree node in level 2 as for example `0.1.0`. It will likely result in some `0.0.`x ($x \in \{0..4\}$) processes actually sending data to `0.1.0`

4.2 Data Distribution Issues

The communication semantics introduced above involves integrating the data model within the message passing framework and results in hiding complexity to programmers. Until now programmers were either obliged to care explicitly about scheduling the multiple low level communications or relying on a support library such as InterComm[8] or KeLP[9]. Our goal is to allow a user to invoke a parallel send/receive operation through the tree structure with a high level view and disregarding distribution details. This facility entails making some modifications to the existing environment and how it manages information.

Support for managing distributed data. We propose to extend the MPI data type with data distribution information, using an API to configure this distribution. The MPI data type is to be improved with the distribution's precise description, including distribution type (blocks, regions, etc.), topology, partitioning information, overlapping areas, etc. Depending on the distribution type, these parameters are subject to modification. The lack of a generic API for setting up the environment with any given distribution leads us to propose different sets of functions specialized in each particular one.

Concretely, in the DISC context, descriptors need to store properties of multidimensional arrays that are hierarchically partitioned in blocks. Neighboring blocks share an overlapping area to exchange data between processes. In order to describe these aspects two new properties are added to MPI data types: *Data Region* and *Interface Region*. Data region contains the distribution's dimensions, size and topological characteristics, and may be a subregion contained within a larger one. In this last case, data structures of both are linked. An operation is also provided to set the mapping of data regions to resources (ie to hrank) so as to make the environment aware of the global distribution. Interface region enables the description of overlapping areas and, hence, describe the behavior of a parallel send/receive of the given distribution. These interface regions can be described as `IN` or `OUT` depending if they are aimed at receiving neighbor's data or at sending contained data to a neighbor. Therefore, the complete description of the distribution is made through a sequence of calls to the mentioned operations. It generates a set of descriptors linked in a tree shape – as a result of the hierarchical partitioning.

As the framework is then aware of the data distribution and how it is assigned to computing resources, it is able to compute the schedule of the various messages needed to fulfill the hierarchical send/receive operations.

If we deepen in this reasoning a new collective operation `update(buf, data_descriptor, hrank}` can be define at the top of this rich information context. Such an operation generates the needed communications to update the overlapping areas. This mechanism strongly simplifies the application programming and makes the code much more legible.

4.3 A Simple Example

Let assume a numerical method has to be applied over a 2D matrix, where data exchanges concern overlapping areas highlighted on Figure 1. Each process sends the part of this area included in the matrix region it manages to the processes managing the neighbor region and receives the other part of the overlapping area. An iteration of the

algorithm's main loop consists of a computing sequence followed by one of these data exchanges and a reduction operation.

As we dispose of a number of resources organized as a hierarchy, we divide hierarchically the matrix relying on the structure and characteristics discovered with the functions presented above. Hence, we provide the framework the necessary information regarding the region partitioning (size), the partition-hrank association and the overlapping areas (beginning, size, hrank that handles the neighbor region, etc) for each level. Once this description has been set, the step's data exchange is done through a call to `update (pointer_to_data, data_descriptor)` that updates the overlapping areas in each process. Except for the extra work of describing data to the framework, no difficulty is added to the programmer's task, while we get in return a simple way to automate a complex send/receive schedule.

4.4 Implementation Issues

On the one hand, the required modifications intend to render the hierarchy of resources visible and easily manageable by application programmers. The necessary changes to prior MPI are related to adequately treating the hierarchy of resources and its associated `hranks`, which requires rich information about resource organization and properties. We should rely, thus, on the services provided by Information Systems specialized in Grid infrastructures that enable us to obtain information needed for managing the proposed environment. For instance, latency and bandwidth may be retrieved from a Network Weather Service [10].

On the other hand, the framework must be able to handle the data that application programmer exchange between processes in a collective way. The work to be done has mostly to do with data distribution issues which has been widely researched as in the InterComm project or in our team's related project PaCO++. Hence, the community has a solid experience on this field.

5 Conclusion

Large scale computing resources are found more and more often organized as hierarchies of heterogeneous machines. MPI implementations has been improved a lot to provide efficient implementation of collective operations on such infrastructures.

However, some parallel applications, dealing with huge arrays and originally designed for a flat message passing model, appear less performing when tested on hierarchically distributed resources. They may be geared up in such execution environment by adapting hierarchical data partitioning, and the communications this implies. For this reason, the hierarchy of resources is to be made accessible and easy to manage within the applications, whereas handling of data distributions is to be supported by the same environment to enhance more efficient communications and code reusability. This is why we propose an MPI extension described in this paper.

The proposed extension allows programming data exchanges and collective operations within the various available levels of a grid environment in an easy way that results directly from the mapping of the resource structure. The complex issue of computing

message exchanges is left to the runtime as well as the responsibilities of computing an efficient message schedule. From the experience of collective operation implementation, the runtime has enough information to achieve it.

The list of function described in this paper is not an exhaustive list. Other functions, like conversion between hierarchical representation and standard MPI communicators are probably needed.

A simple PDE application relying on such an API has been implemented as well as the needed API for this application. While experiments are still to be done, such an API has shows us that it greatly simplifies the management of message exchanges. Such a simplification is gained at the price of requiring a complex hierarchical data partitioning. Libraries are needed to hide this task because it is quite complex to program. Moreover, an extension mechanism should be added in MPI so as to be able to integrate various data redistributions.

Last, we plan to finish the experiments as well as evaluate the benefits of having such concepts in MPI compared to provide them at the level of component models.

References

1. Karonis, N.T., Toonon, B., Foster, I.: MPICH-G2: a Grid-enabled implementation of the Message Passing Interface. J. Parallel. Distrib. Comput. 63, 551–563 (2003)
2. Foster, I., Kesselman, C.: Globus: a metacomputing infrastructure toolkit. Int. J. Supercomput. Appl. 11, 115–128 (1997)
3. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104 (2004)
4. Matsuda, M., Kudoh, T., Kodama, Y., Takano, R., Ishikawa, Y.: Efficient mpi collective operations for clusters in long-and-fast networks. In: IEEE International Conference on Cluster Computing, pp. 1–9. IEEE Computer Society Press, Los Alamitos (2006)
5. Bertrand, F., Bramley, R., Sussman, A., Bernholdt, D.E., Kohl, J.A., Larson, J.W., Damevski, K.B.: Data redistribution and remote method invocation in parallel component architectures. In: IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers, IEEE Computer Society Press, Washington, DC, USA (2005)
6. Pérez, C., Priol, T., Ribes, A.: A parallel corba component model for numerical code coupling. The International Journal of High Performance Computing Applications (IJHPCA) 17, 417–429 (2003)
7. Forum, S.D.R.D.: Document for the data reorganization interface (dri-1.0) (2002), http://www.data-re.org
8. Lee, J.Y., Sussman, A.: Efficient communication between parallel programs with intercomm (2004)
9. Fink, S.S.K., Baden, S.: Efficient runtime support for irregular block-structured applications. J. Parallel. Distrib. Comput. 50(1), 61–82 (1998)
10. Wolski, R., Spring, N., Hayes, J.: The network weather service: A distributed resource performance forecasting service for metacomputing. Journal of Future Generation Computing Systems 15, 757–768 (1999)

MetaLoRaS: A Re-scheduling and Prediction MetaScheduler for Non-dedicated Multiclusters*

J. Ll. Lérida[1], F. Solsona[1], F. Giné[1], M. Hanzich[2], J.R. García[2], and P. Hernández[2]

[1] Departamento de Informática e Ingeniería Industrial, Universitat de Lleida, Spain
{jlerida,francesc,sisco}@diei.udl.es
[2] Departamento de Arquitectura y Sistemas Operativos, Universitat Autònoma de Barcelona, Spain
{mauricio,jrgarcia}@aomail.uab.es, {porfidio.hernandez}@uab.cat

Abstract. Recently, Multicluster environments have become more important in the high-performance computing world. However, less attention has been paid to non-dedicated Multiclusters. We are developing MetaLoRaS, an efficient two-level MetaScheduler for non-dedicated environments, which assigns PVM and MPI applications according to an estimation of the turnaround time in each particular cluster.

The main MetaScheduler goal is to minimize the average job turnaround time in a non-dedicated environment. The efficiency of MetaLoRaS depends on the prediction accuracy of the system and its ability to take decisions according to changes in local workload.

In this paper we present different Metascheduling techniques that take the dynamics of the local workload into account and compare their effects on system performance. We evaluate the prediction accuracy in relation to the low-level queues sizes. Finally, we analyze the relationship between prediction accuracy and system performance.

1 Introduction

We are interested in making use of the wasted computational resources of non-dedicated and heterogeneous Multiclusters to execute parallel applications efficiently. A Multicluster system has a network topology made up of interconnected clusters, limited to a campus- or organization-wide network.

In a Multicluster, the CPU power of the nodes and bandwidth of the interconnection networks are well known. The workload is more stable and can be more precisely estimated than in grid environments. Due to this, Multiclusters can make more accurate predictions of the execution of parallel applications.

Our research interest is in the design of Metaschedulers for Multiclusters. In [7] we presented MetaLoRaS, an efficient Metascheduler made up of a queuing system with two-level hierarchical architecture for non-dedicated Multiclusters. The most important contribution was the effective cluster selection mechanism, based on the estimation of the job turnaround time. Parallel applications are assigned to clusters where the minimum estimated turnaround time is obtained. The conclusions presented by Epema

* This work was supported by the MEyC-Spain under contract TIN 2004-03388.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 195–203, 2007.

and Santoso (in [2] and [9]), corroborate the good selection of the hierarchical model. Epema also showed that the inclusion of schedulers in each cluster guarantees the scalability of the system. Their inclusion was also justified by the efficiency of the scheduling of single-component jobs, the most widely representative kind of user jobs.

The performance of MetaLoRaS was compared with traditional schedulers based on Best Fit, Round Robin, or some variant on these ([13,9,8,2]). Although the MetaLoRaS model obtained good results, it was shown that the estimation accuracy basically depends on the queue lengths in the Cluster systems.

In this paper we present a new proposal, consisting of reducing length of the bottom queues, the ones associated with each cluster. A study of the most suitable configuration depending on the use of the resources and the dynamics of the workload is also performed. By limiting the queues, opportunities arise for new metascheduling mechanisms and favor the proposal of policies with re-scheduling of the jobs. The rescheduling mechanism provides the system with the ability to adjust scheduling decisions to changes in the local workload, which can vary rapidly and continuously in non-dedicated Multiclusters.

The most efficient prediction systems presented in the literature are usually based on advanced resource reservations (such as Nimrod/g and Ursala [3,11]), which try to find time intervals reserved for the execution of distributed applications. Snell showed in [11] how jobs requiring a dedicated start time fragment the scheduler's time space as well as its node space, making it more difficult to assign jobs to idle resources. Consequently, the resources may go unused and cannot be applied to our system.

There are some works on Multicluster rescheduling. A hierarchical space-sharing scheduling with load redistribution between clusters was presented in [1]. In [12] and [8] the distribution of each job to the least loaded clusters in a redundant way was proposed. Although these environments balance very well and improve the system utilization, their implementation requires too many control mechanisms and they do not provide any estimation of the parallel applications execution.

In summary, although the related environments can give acceptable performance, they are not envisaged to provide efficient parallel job performance with high prediction accuracy in a non-dedicated Multicluster, the most important aim of this work.

The remainder of this paper is set out as follows. In Section 2, the Multicluster Scheduling system (MetaLoRaS) is presented. The efficiency measurements of MetaLoRaS are performed in Section 3. Finally, the main conclusions and future work are explained in Section 4.

2 MetaLoRaS

In [7] we proposed a Multicluster architecture. The jobs arriving in the Multicluster enter the Upper-level Queue, awaiting scheduling by the Metascheduler named MetaLoRaS. Next, the Metascheduler assigns the job to the cluster with the minimum estimate of turnaround time by sending the job to the Low-level queue in the selected cluster. The estimation is obtained by each local cluster or Low-level scheduler, named LoRaS (Long Range Scheduler). LoRaS [4], is an space-sharing scheduler, with an efficient turnaround predictor [5], that deals with PVM and MPI parallel applications.

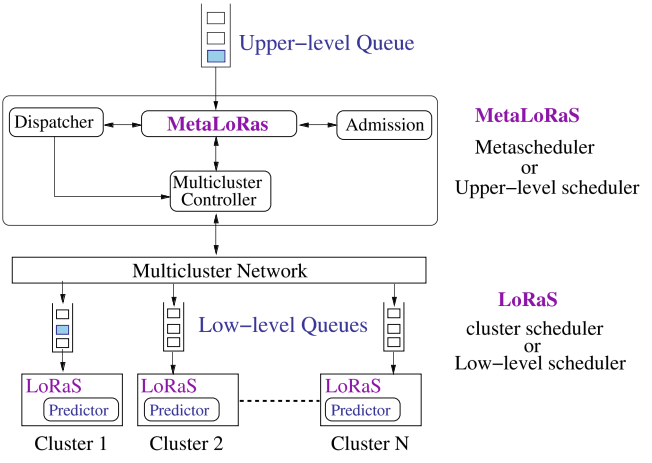


Fig. 1. Multicluster architecture

The MetaLoRaS scheduler has two main issues to resolve: *deviations in prediction accuracy* and low performance in situations with dynamic local workload, due to the *static scheduling*.

The *deviations in the prediction accuracy* arise for two main reasons. MetaLoRaS assigns the jobs to the Cluster with the minimum estimate of turnaround time. To obtain this estimation, each LoRaS (Low-level scheduler) calculates by simulation the remaining execution time of all jobs in its waiting queue, taking into account the use of the resources and the local activity. In this process, the estimation deviation increases with the length of the queues. Furthermore, the estimations do not consider future job arrivals, so it also grows by increasing the number of works that enter the system after each estimation. The last case is outwith the scope of this article, and will be dealt with in a later work.

The other main issue is the *static scheduling*. MetaLoRaS scheduling does not take changes in the system state into account. This means that if the amount of resources or computing power available varies, the jobs cannot be rescheduled. The static nature of the assignments means the system is not sensitive to state changes due to variations in the local workload.

To solve both problems, we propose limiting the number of jobs in the waiting queues of each LoRaS front-end, i.e. their associated Low-level queues. This will improve the effectiveness of the LoRaS predictors, which depend on the number of waiting jobs in their respective Low-level queues.

Once the queues of the local clusters had been limited, we implemented four new job Metascheduling policies. The aim of these new techniques is to detect the most suitable assignments of jobs in the Multicluster. This depends basically on the computing resource requirements and the dynamics of the local activity.

The proposed new policies are the combination of blocking and non-blocking with rescheduling features provided by our MetaScheduler. According to this, the 4 proposed policies are Blocking (B), Non-Blocking (NB), Blocking with Re-Scheduling (B.RS)

and Non-Blocking with Re-Scheduling (NB.RS). Algorithm 1 shows the main steps that MetaLoRaS performs in scheduling jobs. In this algorithm, we select the blocking feature by setting the variable *Blocking*, and if not, the non-blocking feature. Next, the different algorithm features are explained separately.

2.1 Blocking

The *Blocking* (B) feature takes the job to be dispatched from the Upper-level queue in a FCFS way (line 2). The jobs arriving in the system will have to wait for all the previous jobs to be dispatched. This feature requests the estimation of the turnaround time of the job from each Cluster (line 6). When the Low-level queue of the selected Cluster is full, the job is blocked (line 10) in the Upper-level queue until a job in the Low-level queue has been executed. When the Low-level queue is unblocked, MetaLoRaS assigns the job according to the previous estimation (line 4). This feature performs a single estimation per job on the arrival of the job at the top of the Upper-level queue.

Algorithm 1. MetaLoRaS Algorithm

Require:

***MCluster*:** List of clusters with their Low-level queue not full. $MCluster = \{C_i, i = 1..n\}$, where n is the number of Clusters making up the Multicluster.

***UpperLQueue*:** List of the waiting jobs in the Upper-level queue. $UpperLQueue = \{J_i, i = 1..k\}$, where k is the number of jobs in the *Upper-level Q*ueue.

1: **While** ($UpperLQueue \neq \{\phi\}$ **and** $MCluster \neq \{\phi\}$) **do**
2: Select the first job from $UpperLQueue$.
3: **If** (J has a previous estimation of the turnaround time) **then**
4: $C =$ Cluster selected in the previous estimation of the job J.
5: **else**
6: Select Cluster $C = C_i \in MCluster$, with minimum estimated turnaround time for job J
7: **end if**
8: **If** (number of jobs in Low-level Queue of Cluster C is equal to its Low-level Queue limit) **then**
9: **if (Blocking)**
10: Block J in the Upper-level queue until the unblocking of Low-level queue of C
11: **else /* non-blocking */**
12: $UpperLQueue = UpperLQueue - J$. Delete the job of the Available jobs list.
13: $MCluster = MCluster - C$. Delete the Cluster of the Available Clusters list.
14: **end if**
15: **Continue** /* Go to the begin of the while loop*/
16: **end if**
17: Dispatch the job J into Cluster C
18: $UpperLQueue = UpperLQueue - J$
19: **end while**

2.2 Non-blocking

The *Non-Blocking* (NB) feature tries to reduce the waiting time of the jobs in the Upper-level queue introduced by the *B* feature. We want to minimize the negative effects on

the application and system performance. The main steps of the *NB* feature are shown in Algorithm 1, between lines 11 to 14.

The idea is to advance the jobs that are waiting in the Upper-level queue, while there is a job at the top of the queue waiting for a blocked Cluster. The main aim is that when a job is assigned to a blocked cluster, the rest of the jobs waiting in the queue will not be delayed whenever there are available clusters. Thus, we aim to improve performance by reducing the waiting times.

This technique is similar to the backfilling introduced by Feitelson in [10]. Although there are proposals based on traditional backfilling applied to Multicluster systems ([14], [6]), they have only been applied in reservation systems. No solutions for non-dedicated environments and hierarchical queues system are proposed in these works.

When a job *J* is blocked waiting for an unblocked cluster, the job is eliminated (line 12) from the list of waiting jobs (*UpperLQueue*). Also, the selected cluster *C* is eliminated (line 13) from the available clusters list (*MCluster*) and then the Metascheduler goes through the *UpperLQueue* list searching for the next job to be scheduled. This is performed while there are clusters in the *MCluster* list and jobs waiting in the *UppeLQueue* list. When all the clusters are blocked, the jobs in the Upper-level queue will remain waiting to be scheduled. As some clusters is unblocked the available clusters list (*MCluster*) is updated.

2.3 Re-scheduling

One of the most important issues of the above policies is their static assignment. The estimation of a job is only performed once, when the job arrive at the top of the Upper-level queue. Thus, when MetaLoRaS takes decisions, the state of the cluster may have varied since the estimation was obtained, thus producing bad assignments.

The *Re-Scheduling* (RS) feature tries to solve this problem. When a blocked job in the Upper-Level queue is unblocked, the MetaLoRaS requests a new estimation of the turnaround time when it detects changes in the Multicluster state with regard to the last estimation of the job. This can cause a new assignament (job scheduling change or re-scheduling) of the job.

We can apply Re-scheduling to both the Blocking and the Non-Blocking features. In doing so, only line 3 of Algorithm 1 must be replaced with the following line:

3: If (*J* has a previous estimation of the turnaround time **and** the Multicluster state has not changed with regard to the previous estimation) **then**

The main advantage of this feature is that without introducing a great estimation cost, it is possible to deal with dynamic local workload changes, and improve the performance of the system with respect to the previously presented static policies.

3 Experimentation

The whole system was evaluated in a Multicluster made up of 3 identical non-dedicated clusters. Each cluster was made up of eight 3-GHz uniprocessor workstations with 1GB of RAM and 2048 KB of cache, interconnected by a 1-Gigabit network.

In order to carry out the experimentation process, the local and parallel workloads need to be defined. The local workload is represented by a synthetic benchmark (named *local_bench*) that can emulate the usage of 3 different resources: CPU, Memory and Network traffic. The use of these resources was parameterized in a real way. According to the values obtained by collecting the user activity in an open laboratory over a number of weeks, *local_bench* was modeled to use 15% CPU, 35% Memory and a 0,5KB/sec LAN, in each node where it is executed.

The parallel workload was a mixture of PVM and MPI applications, with a total of 60 NAS parallel jobs (CG, IS, MG, BT), with a size of 2, 4 or 8 tasks (class A and B), which reached the system under a Poisson distribution. These jobs were merged so that the entire parallel workload had a balanced requirement for computation and communication. Throughout the experimentation, the maximum number of simultaneous parallel jobs in execution per node was 4.

The performance was also measured by using static and dynamic local workloads. The static workload assigns user jobs to a percentage of Multicluster nodes (0, 25, 50, 75 or 100%) throughout the execution of the parallel workload. However, in the dynamic case the nodes that are loaded vary arbitrarily.

In the experiment, we were interested in knowing the effect of the Low-level queue size on the application performance and the benefits of the new Re-scheduling techniques based on the state of the Multicluster. In order to do this, a comparison was done between different techniques with different Low-level queue sizes. The Low-level queue size was varied in the range from 1 to unlimited (UL). The techniques evaluated were Blocking (B), Non-Blocking (NB), and Blocking and Non-Blocking with Re-Scheduling (B.RS and NB.RS respectively).

In order to be able to evaluate the results two different metrics, *Turnaround Time* and *Prediction Deviation*, were used. The *Turnaround Time* is the waiting plus the execution time of one job, while *Prediction deviation* is a percentage that reflects the difference between the real turnaround time and its estimation.

3.1 Prediction Deviation Analysis

Figure 2 shows the Prediction Deviation for the overall policies when the limit of the cluster queues varied. As we can see, with dynamic load the Prediction Deviation grows more quickly with the length of the Low-level queues than in the static case. This fact means that prediction deviation increase with variability in the local workload.

Moreover, prediction accuracy depends on the length of the Low-level queues. This is because the accumulated error in prediction increases with the number of jobs in the Low-level queues. The rescheduling property favors prediction accuracy because the recent state of the cluster is considered. Rescheduling policies obtained an average gain of 4% in comparison with the non-rescheduling ones.

3.2 Performance Analysis

Figure 3 shows the performance of the overall policies. In general, the increase length of the queues favors application performance because it reduces the waiting time in the Upper-level queue enormously. The rescheduling techniques obtained the best results,

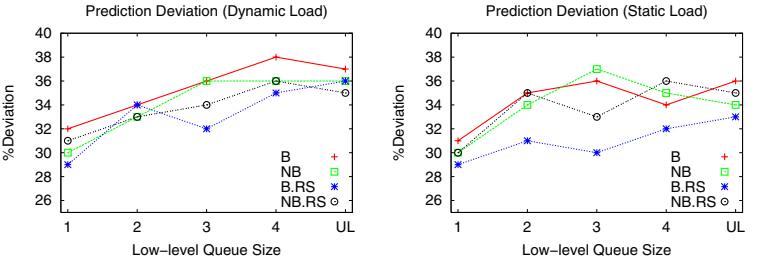


Fig. 2. Prediction Deviation

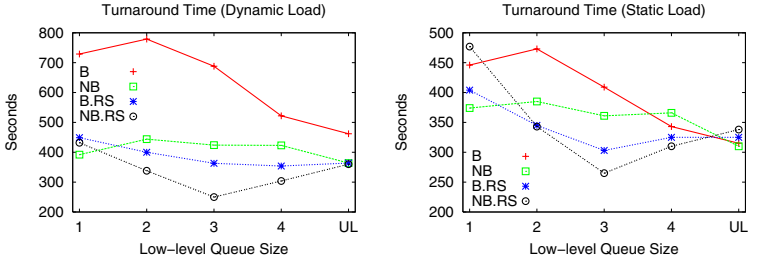


Fig. 3. Average Turnaround Time

because as happened in the deviation case, the system state is more realistic, and even more so in the specific case of the dynamic load. In this case the best performance is obtained by the NB.RS technique with an average gain of 13% compared with the B.RS.

In general, the blocking policies gave worse performance results than the non-blocking ones because the waiting time spent blocked in the Upper-level queue lowered the system and application performance.

As can be seen in Figures 2 and 3 there is an optimal prediction and performance length of the Low-level queues. In our case, this value is 3. This value coincided with the mean Low-level occupancy. This value depends on the computing power of the component clusters and the inter-arrival rate of the jobs.

For values above the optimum, both prediction and performance results tend to stay stable. For long Low-level queue sizes, the waiting time in the Upper-Level queue is zero, then neither the Re-Scheduling nor Blocking Metascheduling techniques have any effect, so that the results of prediction deviation and performance are independent of the Low-level queue size.

4 Conclusions and Future Work

This paper presents new Metascheduling techniques for non-dedicated Multicluster environments that deal with changes in local workload. One of the most important aims of this work is to provide the possibility of defining the most suitable configuration for a Multicluster based on the parallel user needs: accuracy in the prediction and performance improvement.

The results show that there is a direct relationship between the Low-level queue sizes, prediction accuracy and performance. In general, Re-scheduling policies obtained the best performance and prediction accuracy, and more specifically in the dynamic load, the most representative one in non-dedicated Multiclusters. Re-Scheduling policies obtained a performance gain with respect to the Non-Rescheduling of 20%, and a prediction accuracy gain of 4%.

In the future work, we will investigate an analytical model to obtain the optimal value of the Low-level queue sizes by simulating the MetaLoRaS queuing system. In doing so, inter-arrival job times and service times of the clusters must be related with the prediction accuracy. Performance comparison will also be made with the traditional (one-level) backfilling policy.

References

1. Abawajy, J., Dandamudi, S.: Parallel Job Scheduling on Multicluster Computing Systems. In: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'03) (2003)
2. Bucur, A., Epema, D.: Local versus Global Schedulers with Processor Co-allocation in Multicluster Systems. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 184–204. Springer, Heidelberg (2002)
3. Buyya, R., Abramson, D., Giddy, J.: Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. In: The 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000) (2000)
4. Hanzich, M., Giné, F., Hernández, P., Solsona, F., Luque, E.: Time Sharing Scheduling Aproach for PVM Non-Dedicated Clusters. In: Di Martino, B., Kranzlmüller, D., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3666, pp. 379–387. Springer, Heidelberg (2005)
5. Hanzich, M., Giné, F., Hernández, P., Solsona, F., Luque, E.: Using On-The-Fly Simulation For Estimating the Turnaround Time on Non-Dedicated Clusters EuroPar 2006. In: Lecture Notes in Computer Science (2006)
6. Lawson, B.G., Smirni, E.: Multiple-Queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 72–87. Springer, Heidelberg (2002)
7. Ll Lérida, J., Solsona, F., Giné, F., Hanzich, M., Hernández, P., Luque, E.: MetaLoRaS: A Predictable MetaScheduler for Non-Dedicated Multiclusters. In: Guo, M., Yang, L.T., Di Martino, B., Zima, H.P., Dongarra, J., Tang, F. (eds.) ISPA 2006. LNCS, vol. 4330, pp. 630–641. Springer, Heidelberg (2006)
8. Sabin, G., Kettimuthu, R., Rajan, A., Sadayappan, P.: Scheduling of Parallel Jobs in a Heterogeneous Multi-site Environment. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 87–104. Springer, Heidelberg (2003)
9. Santoso, J., van Albada, G.D., Nazief, B.A., Sloot, P.M.: Hierarchical Job Scheduling for Clusters of Workstations. In: Proceedings of the sixth annual conference of the Advanced School for Computing and Imaging (ASCI 2000), pp. 99–105 (2000)
10. Shmueli, E., Feitelson, D.G.: Backlling with lookahead to optimize the performance of parallel job scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 228–251. Springer, Heidelberg (2003)
11. Snell, Q., Clement, M., Jackson, D., Gregory, C.: The Performance Impact of Advance Reservation Meta-scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPDPS-WS 2000 and JSSPP 2000. LNCS, vol. 1911, Springer, Heidelberg (2000)

12. Subramani, V., Kettimuthu, R., Srinivasan, S., Sadayappan, P.: Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (2002)
13. Xu, M.: Effective Metacomputing using LSF MultiCluster ccgrid. In: 1st International Symposium on Cluster Computing and the Grid, p. 100 (2001)
14. Yue, J.: Global Backfilling Scheduling in Multiclusters. In: Manandhar, S., Austin, J., Desai, U., Oyanagi, Y., Talukder, A.K. (eds.) AACC 2004. LNCS, vol. 3285, pp. 232–239. Springer, Heidelberg (2004)

Using CMT in SCTP-Based MPI to Exploit Multiple Interfaces in Cluster Nodes

Brad Penoff[1], Mike Tsai[1], Janardhan Iyengar[2], and Alan Wagner[1]

[1] University of British Columbia, Vancouver, BC, Canada
{penoff,myct,wagner}@cs.ubc.ca
[2] Computer Science, Connecticut College, New London, CT, USA
iyengar@conncoll.edu

Abstract. Many existing clusters use inexpensive Gigabit Ethernet and often have multiple interfaces cards to improve bandwidth and enhance fault tolerance. We investigate the use of Concurrent Multipath Transfer (CMT), an extension to the Stream Control Transmission Protocol (SCTP), to take advantage of multiple network interfaces for use with MPI programs. We evaluate the performance of our system with micro-benchmarks and MPI collective routines. We also compare our method, which employs CMT at the transport layer in the operating system kernel, to existing systems that support multi-railing in the middleware. We discuss performance with respect to bandwidth, latency, congestion control and fault tolerance.

1 Introduction

Clusters are widely used in high performance computing, often with more money invested in processors and memory than in the network. As a result, many clusters use inexpensive commodity Gigabit Ethernet (GbE) rather than relatively expensive fast interconnects[1]. The low cost of GbE cards and switches makes it feasible to equip cluster nodes with more than one interface cards, each interface configured to be on a separate network. Multiple network connections in such an environment can be used to improve performance, especially for parallel jobs which may require more high-bandwidth, low-latency communication and higher reliability than most serial jobs.

We investigate the use of Concurrent Multipath Transfer (CMT) [2,3], an extension to the Stream Control Transmission Protocol (SCTP) [4,5,6], to take advantage of multiple network connections. SCTP is an IETF-standardized, reliable, and TCP-friendly transport protocol, that was initially proposed for telephony signaling applications. It has since evolved into a more general transport protocol for applications that have traditionally used TCP for their requirements of reliability and ordered delivery. SCTP uses the standard sockets library and

[1] In the latest rankings of the Top 500 supercomputers, 211 of the systems use GbE as the internal network [1].

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 204–212, 2007.

an application can interface with SCTP just as with TCP. We have recently released SCTP-based middleware in MPICH2 (`ch3:sctp`) that enables MPI-based parallel programs to use SCTP's features [7].

SCTP introduces several new features at the transport layer such as *multihoming* and *multistreaming* that make it interesting for use in clusters. Of particular relevance to this research, SCTP's multihoming feature allows multiple network interfaces to be used within a single transport layer *association*—a TCP connection can bind to only one interface at each endpoint. In standard SCTP, only a single path is used for data transfers; the alternative paths are only used as backups if the primary path fails. CMT is a recent proposed extension to SCTP that aggregates, at the transport layer, available bandwidth of multiple independent end-to-end paths between such multihomed hosts [3].

Our work is related to projects such as NewMadeleine [8], MVAPICH [9], and Open MPI [10] that support multihomed or multi-railed systems in the middleware. However, these projects focus on low latency network architectures such as Infiniband, and not just IP networks. NewMadeleine and Open MPI both support heterogeneous networks; CMT can be used to combine all IP interfaces, which in turn can be used in combination with these heterogeneous network solutions. MuniCluster [11] and RI2N/UDP [12] support multi-railing for IP networks at the application layer, and use UDP at the transport layer. Our approach is significantly different in that CMT supports multihoming (or multi-railing) at the transport layer[2], as part of the kernel, rather than at the application layer in user-space.

The remainder of the paper is organized as follows. In Section 2, we provide a brief overview of relevant SCTP and CMT features. We then discuss our approach, in comparison with other projects, with respect to configuration, scheduling, congestion control, and fault tolerance. In Section 3, we describe our experiments to evaluate CMT in a cluster for MPI programs, and discuss the performance results. We give our conclusions in Section 4.

2 Issues Related to Using SCTP and CMT in a Cluster

An example of a generic cluster configuration that can benefit from our approach is shown in Figure 1. This cluster has two nodes, each equipped with three Ethernet interfaces. On a node, each interface is connected to one of three IP networks through an Ethernet switch—two interfaces are connected to internal cluster networks, and the third interface is connected to a public and/or management network. In such a configuration, the internal networks can be used in parallel to improve cluster communication in several ways. A system that uses these separate networks simultaneously must also account for possibly varying and different bandwidths and/or delays of the different networks.

In this section, we discuss how SCTP and CMT enable this communication. To compare our approach to existing approaches, we discuss points that any

[2] We also investigated channel bonding, but did not observe any performance advantage in our setup.

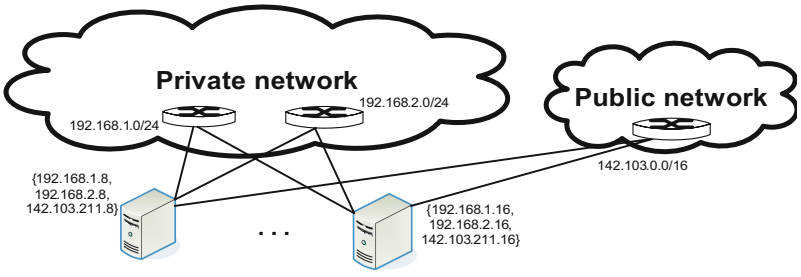


Fig. 1. Example Configuration

multi-railing solution must consider: configuration, scheduling, congestion control and fault tolerance.

2.1 Configuration

An SCTP association between two endpoints can bind to multiple IP addresses (i.e., interfaces) and a port at each endpoint. For example, in Figure 1, it is possible to bind ({192.168.1.8, 192.168.2.8}:5000) at one endpoint of an association to ({192.168.1.16, 192.168.2.16}:4321) at the other endpoint. By default, SCTP binds to all the interfaces at each endpoint, but the special `sctp_bindx()` API call can be used to restrict the association to a subset of the interfaces at each endpoint (i.e., the private network ones). We provide this configuration information to the MPI middleware in a configuration file that is loaded at run-time. This is the only interface configuration information needed by the system. Unlike related projects, the bandwidth and latency characteristics of the network are not provided statically but determined dynamically and adaptively as part of the transport protocol (see Section 2.3).

A final issue with respect to configuration is routing. CMT and SCTP determine only which destination address to send data to; at the transport layer, CMT and SCTP do not determine routes through the network. Routing is done by the routing component of the network layer using the kernel's routing/forwarding tables, and this component determines the local interface to use to get to a specified destination. For example, in Figure 1, the routing table at the host with the public address 142.103.211.8 must be configured so that the local outgoing interface 192.168.1.8 (and neither 192.168.2.8 nor 142.103.211.8) is used when the destination address for an outgoing packet is 192.168.1.16. This kind of routing configuration is straightforward and well understood by system administrators, who perform similar nominal configurations for all IP hosts on the network.

2.2 Scheduling

In our SCTP-based MPI middleware, MPI messages are split into one or more SCTP data *chunks* at the transport layer—data *chunks* are the smallest atomic schedulable unit in SCTP—which are then packed into a transport layer segment

and sent out the primary link, in the case of standard SCTP. However, CMT maintains a congestion window (`cwnd`) for each network path and *distributes* data segments across multiple paths using the corresponding `cwnd`s, which control and reflect the currently allowed sending rate on their corresponding paths. CMT thus distributes load according to the network paths' bandwidths—a network path with higher available bandwidth carries more load.

CMT's scheduling results in a greedy scheduling of data segments to interfaces, which is similar to previous approaches. The main difference is that because CMT schedules data chunks, both short and long MPI messages are sent concurrently over multiple paths, which gives more flexibility in scheduling at a finer granularity than message-level granularity. A disadvantage of our approach is that it is difficult within CMT's scope to identify MPI message boundaries, making it difficult to schedule messages based on message size. For instance, CMT may have difficulty scheduling a small MPI message on the shorter delay path to minimize latency.

2.3 Congestion Control

Most of the existing work on multi-railing assumes dedicated resources on well-provisioned networks and do not consider the effect of congestion on network performance, using specialized protocols that lack congestion control. However, congestion is more likely in clusters with commodity Ethernet switches, especially for the bursty traffic in MPI programs. The problem of congestion is exacerbated in larger networks with multiple layers of switches and mixes of parallel and sequential jobs executing concurrently. In the face of congestion, senders must throttle their sending rate and avoid persistent network degradation.

SCTP, like TCP, uses congestion control mechanisms to minimize loss and retransmissions and to ensure that application flows are given a fair share of the network bandwidth. SCTP dynamically tracks a `cwnd` per destination address, which determines the current sending rate of the sender (recall that the source address is chosen by the routing component of the network layer). CMT extends SCTP's mechanisms to enable correct congestion control when sending data concurrently over multiple paths. Our system thus differs significantly from the previous approaches in that CMT is dynamic, not static, and will correctly adjust its sending rate to changing traffic characteristics in the network. CMT congestion control mechanisms ensure that (i) when a new flow enters the network, existing flows curb their `cwnd`s so that each flow gets a fair share of the network bandwidth, (ii) when flows leave the network, continuing flows adapt and use the spare network bandwidth, and (iii) CMT can be used over network paths with differing bandwidths/latencies.

2.4 Fault Tolerance

Fault tolerance is an important issue in commodity cluster environments, where network failure due to switches or software (mis)configuration does occur. SCTP has a mechanism for automatic failover in the case of link or path failure. Similar

mechanisms have been implemented in MPI middleware [13], but these are unnecessary for SCTP-based middleware. SCTP uses special control chunks which act as *heartbeat* probes to test network reachability. Heartbeat frequency is user-controllable, and can be tuned according to expectations for a given network. However, at the currently recommended setting [14], it takes approximately 15 seconds to failover—a long time for an MPI application. Since CMT sends data to all destinations concurrently, a CMT sender has more information about *all* network paths, and can leverage this information to detect network failures sooner. CMT is thus able to reduce network path failure detection time to one second [15]. This improvement with CMT is significant, and future research will investigate if this time can be further reduced for cluster environments.

3 Performance

The goal of our evaluation was to see that CMT could take advantage of the available bandwidth, scheduling within the transport protocol. The focus of the evaluation was on bandwidth intensive MPI benchmarks. The hope was to demonstrate results comparable to MPI implementations that schedule within the middleware.

Test Setup. The potential benefits of using CMT with MPI were evaluated on four[3] 3.2 GHz Pentium-4 processors in an IBM eServer x306 cluster, where each node had three GbE interfaces. The two private interfaces were on separate VLANs connected to a common Baystack 5510 48T switch, and the third (public) interface was connected to a separate switch, similar to the setup shown in Figure 1. We tested with an MTU size of 1500 bytes and with jumbo frames of size 9000 bytes.

For the tests with SCTP and CMT we used MPICH2 v1.0.5, which contains our SCTP-based MPI channel (`ch3:sctp`). We tested MPICH2 with and without CMT enabled and compared it to TCP from the same release of MPICH2 (`ch3:sock`). We also tested Open MPI v1.2.1, which supports message striping across multiple IP interfaces [16]. Open MPI was ran with one and two TCP Byte-Transfer-Layer (BTL) modules to provide a comparison between using one interface versus two interfaces[4].

All SCTP tests were based on an SCTP-capable stack[5] in FreeBSD 6.2. For all tests we used a socket buffer size of 233 Kbytes, the default size used by `ch3:sctp`, and we set the MPI rendezvous threshold for MPI to 64 Kbytes, matching `ch3:sctp` and Open MPI's default.

[3] We did not have the opportunity to test our implementation in a large cluster. Since SCTP was designed as a standard TCP-like transport protocol, we expect its scalability to be similar to that of TCP.

[4] Our network paths were not identical so Open MPI had the network characteristics set in the `mca-params.conf` file.

[5] A patched version of the stack (`http://www.sctp.org`) was used. SCTP will be a standard part of FreeBSD 7– as a kernel option.

Microbenchmark Performance. The most important test was to determine whether MPI with CMT was able to exploit the available bandwidth from multiple interfaces. We used the OSU MPI microbenchmarks [17] to test bandwidth and latency with varying message sizes. We tested with MTUs of 1500 and 9000 bytes and found that using a 9000 byte MTU improved bandwidth utilization for both TCP and SCTP. Figure 2 reports our results with MTU 9000.

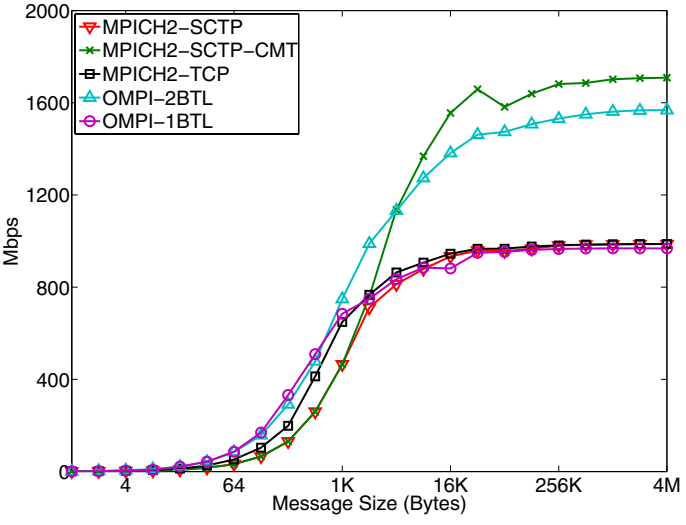


Fig. 2. OSU Bandwidth Test comparing MPICH2 with and without CMT and Open MPI with one and two TCP BTLs

As shown in Figure 2, MPICH2-`ch3:sctp` with CMT starts to take advantage of the additional bandwidth when the message size exceeds 2 Kbytes. At a message size of 1 Mbyte, MPICH2-`ch3:sctp` with CMT is almost achieving twice the bandwidth (1.7 versus 0.98 Gbps) in comparison to using one interface, and it is outperforming Open MPI (1.56 Gbps) which is scheduling messages in user-space across two interfaces within its middleware. All single path configurations are approximately the same, with a maximum of 0.98 Gbps; however for configurations using two networks, Open MPI is the best for message sizes 512 to 2048 bytes but once message size exceeds 4 Kbytes MPICH2-`ch3:sctp` with CMT is the best performing.

In the case of the OSU latency benchmark (figure not shown), `ch3:sctp` without CMT was between `ch3:sock`, which had the lowest latency at 77μsec (for zero-byte message), and Open MPI (one TCP BTL), which was consistently 20% higher than `ch3:sock` (for message size up to 1 Kbytes). Large message latency was more or less the same across all of these middleware configurations that used a single interface. When using two interfaces, CMT increased message latency by approximately 2.5% over `ch3:sctp` for small messages, but this increase was

smaller than the 10% increase we found with Open MPI with two TCP BTLs compared to Open MPI with one TCP BTL. We note that CMT is currently optimized for bandwidth only, and we have not yet explored the possibility of optimizing CMT for both bandwidth and latency.

The third component of performance that we investigated was CPU utilization. For any transport protocol there are trade-offs between bandwidth and the associated CPU overhead due to protocol processing, interrupts, and system calls. CPU overhead is important since it reduces the CPU cycles available to the application and reduces the potential for overlapping computation with communication. We used `iperf`, a standard Unix tool for testing bandwidth, together with `iostat` to investigate CPU utilization differences. CPU utilization for TCP and SCTP with an MTU of 1500 bytes were similar. However, with CMT, the CPU was a limiting factor in our ability to fully utilize the bandwidth of both GbE interfaces. For an MTU of 9000 bytes, the CPU was still saturated, yet CMT was able to achieve bandwidths similar to those obtained from the OSU bandwidth benchmark because more data could be processed with each interrupt. We conclude from our investigation that there is an added cost to protocol processing for CMT. Part of this cost may be due to the cost of processing *selective acknowledgement (SACK)* blocks at the transport layer receiver, which is higher in CMT. SACK processing overhead is actively being considered by developers in the SCTP community, and continued work should reduce this overhead.

Collective Benchmarks. The benchmarks in the previous section tested the effect of CMT and MPI point-to-point messaging on raw communication performance. We were interested in testing CMT's effect on more complex MPI calls like the collectives since they are also commonly used in MPI programs. The Intel MPI Benchmark suite 2.3 was used in our experiments. We tested several of the collective routines, but for brevity, we focus on the `MPI_Alltoall` four process results here since the results obtained for the other collectives showed similar benefits.

The results are summarized in Figure 3; only the multi-rail MPI results are shown (i.e., `ch3:sctp` with CMT and Open MPI with two TCP BTLs). Latency in the Alltoall test is always lower with CMT, except for message sizes of 64 Kbytes and 128 Kbytes. For large messages, CMT can have a considerable advantage over two TCP BTLs (e.g. 42% for 1 MByte messages). Although not pictured here, we note that without multi-railing, MPICH2 (both `ch3:sctp` and `ch3:sock`) is generally within 10-15% of Open MPI with one BTL. Multi-railing starts to show an advantage for message sizes greater than 8 Kbytes. We attribute the performance improvement of MPICH2-`ch3:sctp` with CMT over Open MPI with two TCP BTLs (up to 42%) to the advantages of scheduling data in the transport layer rather than in the middleware. We are currently implementing an SCTP BTL module for Open MPI that will be able to use CMT as an alternative for striping, and we hope to obtain similar performance improvements in Open MPI for multiple interfaces.

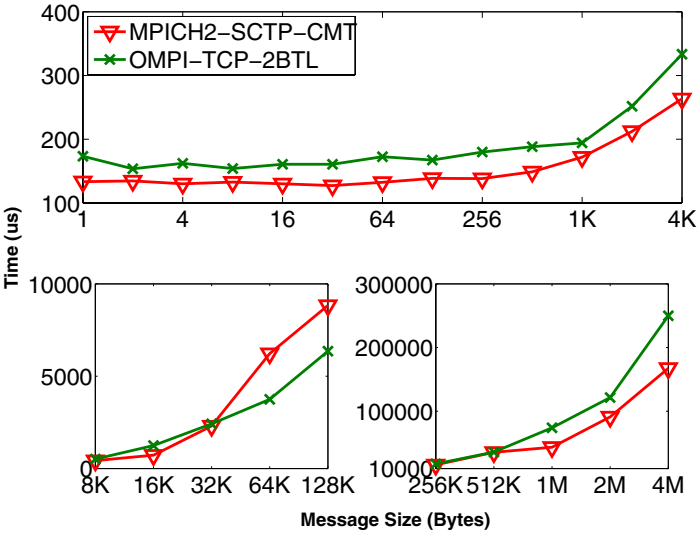


Fig. 3. Pallas Alltoall n=4 Benchmark MTU 1500

4 Conclusions

We have investigated a novel approach that uses multiple network interfaces on cluster nodes to improve the performance of MPI programs. Unlike systems that support multi-railing in the middleware, our approach uses CMT, an extension to SCTP, that supports multi-railing at the transport layer in the kernel. CMT dynamically adapts to changing network conditions, and therefore simplifies the network configuration process. We showed that CMT is able take advantage of the additional bandwidth of an additional interface, and compares well with Open MPI's load balancing scheme. Experiments with MPI collectives demonstrate the possible advantages in scheduling messages at the transport layer (using CMT) versus scheduling messages in the middleware (using Open MPI).

CMT shows promise for bandwidth intensive applications. Future work will investigate optimization of CMT scheduling for lower latency and CMT refinements for fault tolerance. We expect that our research will lead to an easy-to-use system for efficiently using multiple interfaces in clusters with low-cost commodity networking.

Acknowledgments. We wish to thank Randall Stewart and Michael Tüxen for their extensive help with the experiments.

References

1. University of Mannheim, University of Tennessee, NERSC/LBNL: Top 500 Computer Sites (2007), http://www.top500.org/
2. Iyengar, J.: End-to-end Concurrent Multipath Transfer Using Transport Layer Multihoming. PhD thesis, Computer Science Dept. University of Delaware (2006)

3. Iyengar, J., Amer, P., Stewart, R.: Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths. IEEE/ACM Transactions on Networking 14(5), 951–964 (2006)
4. Stewart, R.R., Xie, Q.: Stream Control Transmission Protocol (SCTP): A Reference Guide. Addison-Wesley Longman Publishing Co. Inc. Reading (2002)
5. Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I.M.K., Zhang, L., Paxson, V.: The Stream Control Transmission Protocol (SCTP) (2000) Available from http://www.ietf.org/rfc/rfc2960.txt
6. Stewart, R., Arias-Rodriguez, I., Poon, K., Caro, A., Tuexen, M.: Stream Control Transmission Protocol (SCTP) Specification Errata and Issues (2006), Available from http://www.ietf.org/rfc/rfc4460.txt
7. Kamal, H., Penoff, B., Wagner, A.: SCTP versus TCP for MPI. In: Supercomputing '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, IEEE Computer Society, Washington, DC, USA (2005)
8. Aumage, O., Brunet, E., Mercier, G., Namyst, R.: High-Performance Multi-Rail Support with the NewMadeleine Communication Library. In: The Sixteenth International Heterogeneity in Computing Workshop (HCW 2007), workshop held in conjunction with IPDPS 2007 (2007)
9. Liu, J., Vishnu, A., Panda, D.K.: Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, p. 33. IEEE Computer Society, Washington, DC, USA (2004)
10. Gabriel, E., et al.: Open MPI: Goals, concept and design of a next generation MPI implementation. In: Proc. 11th EuroPVM/MPI, Budapest, Hungary (2004)
11. Mohamed, N., Al-Jaroodi, J., Jiang, H., Swanson, D.R.: High-performance message striping over reliable transport protocols. The Journal of Supercomputing 38(3), 261–278 (2006)
12. Okamoto, T., Miura, S., Boku, T., Sato, M., Takahashi, D.: RI2N/UDP: High bandwidth and fault-tolerant network for PC-cluster based on multi-link Ethernet. In: 21st International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, Workshop on Communication Architecture for Clusters (2007)
13. Vishnu, A., Gupta, P., Mamidala, A.R., Panda, D.K.: Scalable systems software - a software based approach for providing network fault tolerance in clusters with uDAPL interface: MPI level design and performance evaluation. In: SC, p. 85 (2006)
14. Caro, A.: End-to-End Fault Tolerance Using Transport Layer Multihoming. PhD thesis, Computer Science Dept. University of Delaware (2005)
15. Natarajan, P., Iyengar, J., Amer, P.D., Stewart, R.: Concurrent multipath transfer using transport layer multihoming: Performance under network failures. In: MILCOM, Washington, DC, USA (2006)
16. Woodall, T., et al.: Open MPI's TEG point-to-point communications methodology: Comparison to existing implementations. In: Proc. 11th EuroPVM/MPI, Budapest, Hungary, pp. 105–111 (2004)
17. Ohio State University: OSU MPI Benchmarks (2007) http://mvapich.cse.ohio-state.edu

Analysis of the MPI-IO Optimization Levels with the PIOViz Jumpshot Enhancement

Thomas Ludwig, Stephan Krempel, Michael Kuhn, Julian Kunkel, and Christian Lohse

Ruprecht-Karls-Universität Heidelberg
Im Neuenheimer Feld 348, 69120 Heidelberg, Germany
t.ludwig@computer.org
http://pvs.informatik.uni-heidelberg.de/

Abstract. With MPI-IO we see various alternatives for programming file I/O. The overall program performance depends on many different factors. A new trace analysis environment provides deeper insight into the client/server behavior and visualizes events of both process types. We investigate the influence of making independent vs. collective calls together with access to contiguous and non-contiguous data regions in our MPI-IO program. Combined client and server traces exhibit reasons for observed I/O performance.

Keywords: Parallel I/O, MPI-IO, Performance Visualization, Trace-Based Tools.

1 Introduction

More and more applications in the field of high performance computing require also high performance file input/output [8]. Some use POSIX I/O with sequential programs and many others already perform parallel I/O from within parallel programs. They deploy MPI for message passing and MPI-IO for the file operations. MPI-IO, which is an integral part of the MPI-2 standard definition, provides many options to actually program the I/O operations. Most implementations of MPI use ROMIO or some derivative of ROMIO to handle the I/O part of the parallel programs. ROMIO includes complex optimization algorithms, namely data sieving and the two-phase protocol.

Currently application programmers run their parallel programs that include I/O, they get some performance values in the form of overall program completion time and other than that have no insight into why the I/O was fast or slow. They have no information on how the performance values depend on I/O server activities being triggered by their client I/O calls and transformed by the middleware layers, i.e. ROMIO and the parallel file system.

This is where we started our project of the Parallel I/O Visualizer PIOViz. It enhances existing tools in the MPICH2 source code distribution [10] and is provided for the PVFS I/O infrastructure [17]. Using PIOViz we evaluated the performance characteristics of a set of MPI-IO options.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 213–222, 2007.

The remainder of this paper is structured as follows: Section 2 will describe related work and the state-of-the-art. Section 3 will explain the concepts of PIOViz followed by section 4 that gives details on the test scenario. Section 5 will discuss results.

2 Related Work and State-of-the-Art

With parallel computing we find two classes of tool concepts which are used for different purposes: First there are on-line tools. They use a monitoring system to get data out of the running application and/or instrument this application. Data is immediately used for several purposes: we can either display them (e.g. with a performance analyzer) or use them for controlling the application (e.g. with a load balancer). Usually data is not stored. A well known representative of this class is Paradyn [12,9]. Second, we have off-line tools. They present data after program completion (or with a considerable delay during program run). In order to do so the monitoring systems and instrumentations write event traces to files that are used afterwards as data base. Representatives are TAU [15,14], the Intel Trace Analyzer [3], Vampir [4], and Jumpshot [13]. Our work is for the family of trace-based tools, in particular for performance analysis. So we will concentrate on these aspects here.

With performance analysis tools (on- and off-line) there was considerable progress in the last recent years. In particular we see a focus towards automatic detection of bottlenecks. The working group Apart (Automatic Performance Analysis: Real Tools) [1] has investigated this issue in depth and several tool enhancements were developed over the years. Visualization gets more sophisticated and several tools allow to compare traces from different program runs. We see also other sources of information being integrated into the trace: e.g. TAU enters events from performance counters provided by the operating system.

What was still missing are two things: An explicit tracing and visualization of the I/O system's behavior and a correlation of program events and system events. We refer to these features as multi-source tracing and semantical trace merging. Our project PIOViz (Parallel I/O Visualizer) fills in this gap [7]. It is a set of tools to be used together with the MPE/Jumpshot environment provided by MPICH2 and with the PVFS environment.

3 PIOViz Concepts

The PIOViz project goal can be described as follows: In order to have a combined view onto client and server activities we must be able to get traces during program runtime from both of them. In order to see the server activities induced by MPI-IO calls we must related the two traces. Visually this will be done by adding arrows between the events of clients and servers. Technically we will merge the two traces before arrow integration. In order to visualize client and server traces at the same time we must have an appropriate tool.

Figure 1 shows the block structure of our concept. MPI client processes which are linked to the PVFS client library get already traced by the MPICH2/MPE environment. We now need to have traces for the servers, too. We will use a similar approach as for the clients (described below). In both case we get at first clog2 node local traces that are merged to slog2 traces for all clients and all servers respectively. Note that the timelines in the two trace files are already synchronized by the MPE environment. These two traces now get merged and visualized concurrently in the Jumpshot tool. The list of issues to be covered in order to visualize this double trace with arrows comprises the following points:

- Generate a single trace from the server processes.
- Forward information from the clients to the servers that allow to find corresponding MPI-IO and Trove calls later on.
- Merge the two traces from all clients and all servers. Timelines are adjusted such that clock shift is compensated.
- Add arrow elements to the combined trace.
- Distribute client and server activities over time lines in the visualizer.

The implementation of our enhancements takes places at various locations: We need to instrument the PVFS server code and add own code to the client library [18]. For the two slog2 trace files we have several tools implemented to manipulate them. As Jumpshot just visualizes time lines without knowing anything about their meaning and context we could even live without any modifications of this tool. However, for better adaptation to our project goals we slightly enhance the GUI. Details on the concepts can be found in [7,5,11,19].

4 Test Scenario

In this section we will discuss the hardware and software environment for our investigations. Further we make assumptions about the expected overall I/O performance to be observed with parallel MPI-IO programs.

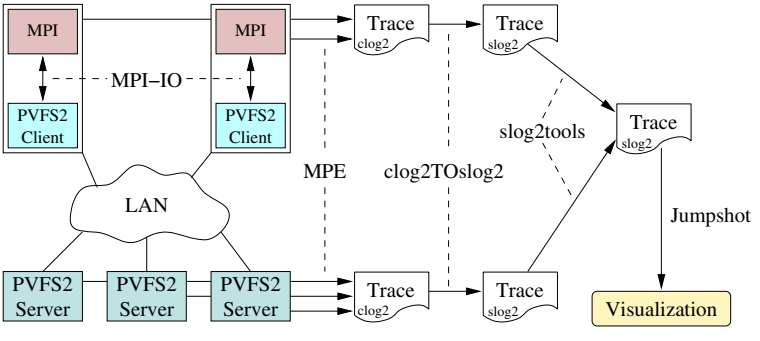


Fig. 1. Architecture of the PIOViz tool environment

This research is conducted on a small cluster consisting of ten nodes, where one node serves as a master. We use the worker nodes for our tests and specify a configuration with four client processes and four I/O server processes being located on different nodes. The nodes have two Xeon processors with one gigabyte of main memory and a gigabit network link. Disks attached to the nodes have a read/write performance of approximately 40 MByte/s. All nodes are interconnected via a switch in a star topology, where the switch has a maximum throughput of 380 MByte/s.

The MPI client processes read and write adjustable amounts of data from and to the I/O servers. The data is transferred either via ten iterative calls, each accessing a contiguous data region or via one single call that accesses a non-contiguous data region with ten data segments being separated by nine holes.

We evaluate the four different levels of access provided to the clients by MPI. They offer varying potential to the underlying software components to optimize the file I/O. In particular ROMIO plays an important role here (for details on ROMIO see [16]). We distinguish the following levels of access complexity:

- level 0: the clients perform ten individual (non-collective) calls accessing one contiguous data region each.
- level 1: the clients do the same as with level 0 but invoke ten collective calls.
- level 2: the clients perform one single individual call accessing one non-contiguous data region
- level 3: the clients do the same as with level 2 but invoke a collective call.

For a detailed discussion of these levels and the appropriate source code see an MPI-2 User's Guide (e.g. [2]).

So what do we expect to see? Level 0 offers no meta information to the underlying software components, in particular to ROMIO. No optimization can take place. Level 1 could perhaps benefit from the fact that it is a collective call. The library is aware of all the processes which are taking part in this particular call and thus could manage things differently. With level 2, which makes access to non-contiguous data regions, ROMIO could apply data sieving mechanisms. Level 3 finally could benefit from the two-phase protocol of ROMIO. With reading, clients just fetch a defined amout of contiguous data and send parts of it via low-level message passing to those clients that need to have this data. Thus, we would expect increasing performance with increasing I/O operation levels.

For these four levels we ran experiments with varying data volumes. Let us at first discuss the effects that we expect to see with respect to overall performance, i.e. completion times of these operations.

The crucial question is: Where is the bottleneck? We can identify different components to become bottlenecks in our scenario under various configurations: Disks have limited read/write throughput, the network links to and from clients and servers have limited throughput and the same is true for the network switch. For simplicity we assume that CPU and the other components of the node's motherboard, which link network interfaces to disk subsystems, do not impose further limitations. Servers and clients in our experiments always run on disjoint

nodes. There is a non-neglectable CPU usage but no interference between these processes. All listed hardware components have also latency time when accessing them. A detailed modelling of bottlenecks can be found in [6].

With respect to different volumes of data we observe the following: As long as the amount of data is low, i.e. up to single megabytes, the overall performance is dominated by the latencies of the participating components. There will be perfect caching of data in the server nodes and disk performance will not play any role here. With an increasing amount of data, where the volume is not yet in the range of the size of the nodes' physical main memory, the overall performance is determined by the maximum network and switch throughput. Disk performance does still not play any role as the data gets cached in the operating system's I/O buffer in main memory. With data volumes exceeding the I/O buffer we will finally enforce physical I/O operations and physical I/O might become the bottlenecks of the chosen configuration. The Linux kernel applies a write-behind strategy, which defers physical write operations. This boosts performance for an amount of data exceeding the cache.

In the following we will present an experiment with medium data volume; thus, we will basically evaluate the performance of clients and servers on which the limitations of the disk subsystem have no further influence on the experiment. Each client transfers a total amount of 50 MByte to the parallel file system (either in a single non-contiguous call or ten contiguous calls). This results in 200 MByte being transferred through the switch from clients to servers. With a maximum throughput of 380 MByte/s we expect a minimum transfer time of 0.53 s. As the Gigabit Ethernet links transfer approximately 120 MByte/s this would result in 0.42 s for the 50 MByte data transfer. Thus, the bottleneck here will be the switch, not the individual links.

In the next section we will discuss the results for this experiment.

5 Discussion of Results

Figures 2–5 visualize the traces obtained by four experiments. They show client and server activities and the cooperation between these processes. We give the traces for write operations — reading exposed an almost identical behavior with respect to performance.

The figures are screen dumps of Jumpshot's main window. The four upper lines (numbered from "0" to "3") visualize the client processes. This is the result obtained using the MPE environment included in MPICH2 and with the shipped Jumpshot. All further time-lines are added by our PIOViz environment and visualize the PVFS servers' behavior. Line "100" refers to the PVFS metadata server. There is only activity when the experiment starts and stops. As the traces here do not show the file open and file close operations there is hardly any communication with the metadata server.

Lines "101" to "104" are the timelines of the four data server processes. A detailed description of the observed behavior is beyond the scope of the paper, but let us make some comments: The green blocks refer to job events in the

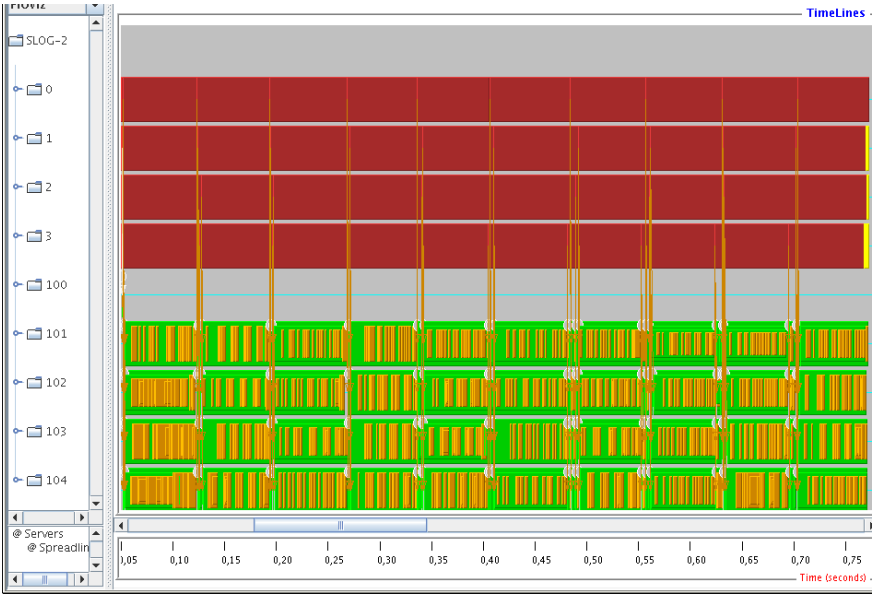


Fig. 2. 4 clients write 50 MBytes each to 4 servers with non-collective calls and contiguous data regions

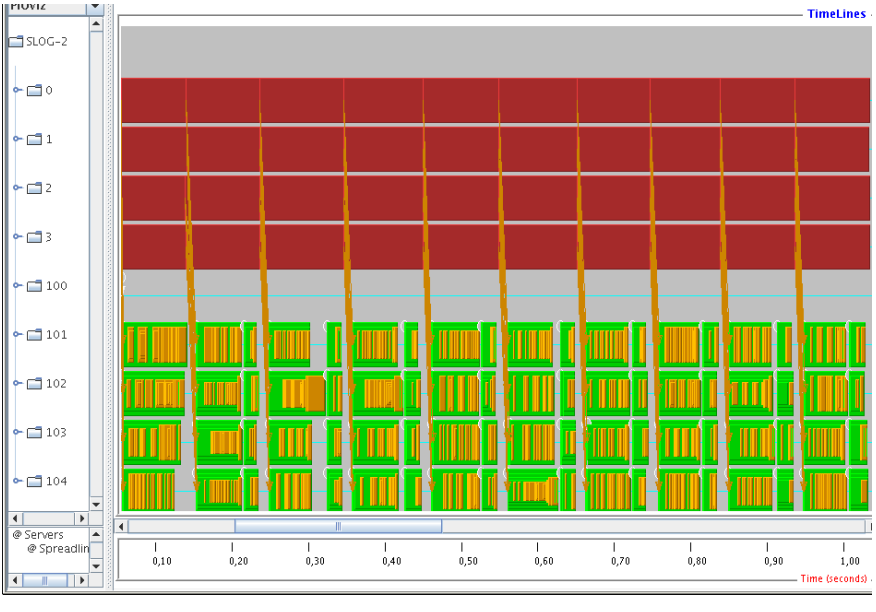


Fig. 3. 4 clients write 50 MBytes each to 4 servers with collective calls and contiguous data regions

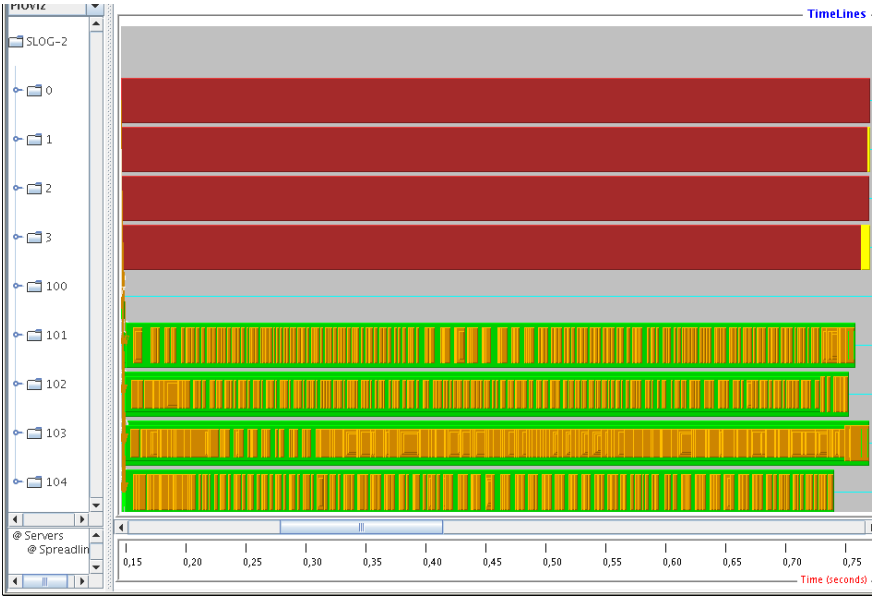


Fig. 4. 4 clients write 50 MBytes each to 4 servers with non-collective calls and non-contiguous data regions

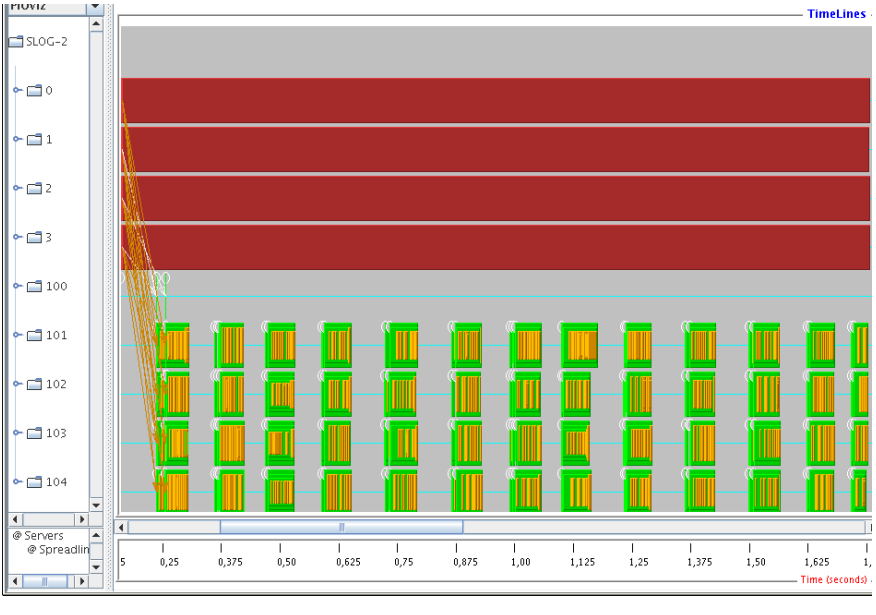


Fig. 5. 4 clients write 50 MBytes each to 4 servers with collective calls and non-contiguous data regions

corresponding PVFS software layer. Jobs are responsible for managing the execution of operations in lower layers (network and persistency layer). The brown segments refer to Trove write operations. Trove is the PVFS software layer that performs the actual disk transfer. Thus, Trove events are related to disk activity (Unfortunately not necessarily directly related, as buffering activities of the other layers might take place, too). There might be an interlaced execution of individual Trove activities resulting in overlapping Trove events on the timelines. We integrated a mechanisms into our environment that allows to expand a single timeline to a set of timelines without overlapping events between different clients (marked by the folder sign left to the process number; not expanded in these screen dumps here).

Further, arrows from client events to server events are incorporated. This is the most important part of PIOViz as it allows to see the causal relation between activities in client and server processes. This contributes to a better understanding of performance related issues in this complex client-server environment.

Remember that with the standard Jumpshot environment from MPICH2 you would get in each of the four traces just the four upper lines from the client processes. They would definitely not give any insight into what is going on in our environment.

Before going into details with the traces let us have a look at the completion times of the different I/O operations. Table 1 shows the measured values for reading and writing and the tested four MPI-IO levels. Reading and writing essentially behaved identically for this experiment (which is slightly different for other experiments). Here we will discuss writing in detail.

With level 0 in fig. 2 we measure a completion time of 0.72 s for the write operation. A lower bound induced by the network would be 0.53 s. The trace shows clearly the ten MPI file write operations that trigger activity in the servers. A more detailed analysis would show that each write operation transfers 5 MByte of data. Due to the RAID-0 striping concept of PVFS each server receives 1.25 MBytes. It uses five Trove write calls to transfer them to the local disks (default transfer size is 256 KByte).

The trace for level 1 in fig. 3 shows a higher completion time where we expected to see a shorter one. Looking at the arrows starting from the write calls at the MPI level we can understand the effect: With level 1 we have a global synchronization after each write call. Thus, the next write call starts well synchronized but also later, because here we always have to wait for the slowest I/O server. With level 0 there is no synchronization and slight differences in single operations are compensated statistically. The server trace of level 1 shows small

Table 1. Completion times of I/O operations with four different concepts of MPI-IO

	level 0 individual contiguous	level 1 collective contiguous	level 2 individual non-contiguous	level 3 collective non-contiguous
write	0.72 s	0.97 s	0.62 s	1.59 s
read	0.77 s	0.94 s	0.54 s	1.57 s

gaps between the Job events, indicating server inactivity during certain periods of time.

Level 2 (refer to fig. 4), which is a non-collective call to a non-contiguous data region has the best overall performance. We see on the left one single cooperation between clients and servers; this is when writing starts. Each client transfers a piece of the data to the appropriate servers. No data sieving takes place in this experiment. As sieving does not trigger events we cannot comment on this at the moment. The completion time is about 100 ms shorter than with level 0. Considering the fact that we now save nine MPI calls with their respective additional latencies this might explain the measured time difference (9 calls to 4 servers each with a round-trip time of about 2 ms would last about 72 ms).

Level 3 in fig. 5 finally, although expected to be the most appropriate for optimizations, yields the worst results. Looking at the trace we see different things: First, the clients show one single call only, where you cannot detect any details. Second, the servers exhibit a cyclic behavior of inactivity followed by a well synchronized block of write events. This is due to the two-phase optimization protocol in ROMIO. It first exchanges data to be written between processes such that they have a contiguous chunk of data (phase one). It uses low level message passing calls for this purpose. We cannot see them in forms of events at the client level as there is no instrumentation available in the library for these calls. However, the inactivity at the servers reveals it. The data is now written by the clients as a contiguous chunk to disk (phase two). As ROMIO uses by default an internal 4 MByte buffer and each client transfers 50 MByte we see 13 blocks of write operations at each server. The last block writes 2 MByte to disk, all others 4 MByte. This inefficiency is caused by a combination of buffer size, data access pattern, and network speed. Further investigations have to be conducted here.

6 Conclusion and Future Work

The PIOViz environment supports the combined tracing of MPI client processes and PVFS I/O server processes. Traces are merged together, adjusted with respect to timing information, and annotated to present causal relations between client and server activities.

We conducted experiments with MPI-IO programs to analyse the effects of different programming concepts onto overall performance of these parallel I/O programs. We investigated individual and collective MPI calls accessing contiguous and non-contiguous data regions. The resulting four levels of I/O are handled differently by optimization algorithms in the ROMIO layer.

The traces of our PIOViz environment show many details about the client-server cooperation that are not at all visible with the regular Jumpshot tool. Many performance issues can now be well explained and understood. This new knowledge will lead to better I/O optimization mechanisms in the future.

Our next steps in this project will be to integrate also performance counter values into the traces and thus get even more insight into CPU, disk, and network

performance issues. The tools will also be applied to real applications to support the tuning of their I/O concepts.

References

1. APART (Homepage), http://www.kfa-juelich.de/apart/
2. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2 — Advanced Features of the Message-Passing Interface. The MIT Press, Cambridge (1999)
3. Intel Trace Analyzer & Collector (Home page), http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/
4. Vampir (Home page), http://www.vampir.eu/
5. Krempel, S.: Tracing Connections Between MPI Calls and Resulting PVFS2 Disk Operations, Bachelor's Thesis, Ruprecht-Karls-Universität Heidelberg, Germany (2006)
6. Kunkel, J., Ludwig, T.: Performance Evaluation of the PVFS2 Architecture. In: Proceedings of the PDP, Naples, Italy (2007)
7. Ludwig, T., et al.: Tracing the MPI-IO Calls' Disk Accesses. In: European PVM/MPI User's Group Meeting, Bonn, Germany, pp. 322–330. Springer, Berlin (2006)
8. Ludwig, T.: Research Trends in High Performance Parallel Input/Output for Cluster Environments. In: Proceedings of the 4th International Scientific and Practical Conference on Programming UkrPROG'2004, National Academy of Sciences of Ukraine, Kiev, Ukraine pp. 274–281 (2004)
9. Miller, B.P., et al.: The Paradyn Parallel Performance Measurement Tool. IEEE Computer. Special issue on performance evaluation tools for parallel and distributed computer systems 28(11), 37–46 (1995)
10. MPICH2 home page (Home page). http://www-unix.mcs.anl.gov/mpi/mpich2/index.htm
11. Panse, F.: Extended Tracing Capabilities and Optimization of the PVFS2 Event Logging Management, Diploma Thesis, Ruprecht-Karls-Universität Heidelberg, Germany (2006)
12. Paradyn Parallel Performance Tools (Home page), http://www.paradyn.org/index.html
13. Performance Visualization for Parallel Programs (Home page), http://www-unix.mcs.anl.gov/perfvis/
14. Shende, S., Malony, A.: The Tau Parallel Performance System. International Journal of High Performance Computing Applications 20, 287–311 (2006)
15. TAU – Tuning and Analysis Utilities (Home page), http://www.cs.uoregon.edu/research/tau/
16. Thakur, R., Lusk, E., Gropp, W.: Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised (July 1998)
17. The Parallel Virtual File System – Version 2 (Home page), http://www.pvfs.org/pvfs2/
18. The PVFS2 Development Team: PVFS2 Internal Documentation included in the source code package (2006)
19. Withanage, D.: Performance Visualization for the PVFS2 Environment, Bachelor's Thesis, Ruprecht-Karls-Universität Heidelberg, Germany (November 2005)

Extending the MPI-2 Generalized Request Interface

Robert Latham, William Gropp, Robert Ross, and Rajeev Thakur

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{robl,gropp,rross,thakur}@mcs.anl.gov

Abstract. The MPI-2 standard added a new feature to MPI called *generalized requests*. Generalized requests allow users to add new nonblocking operations to MPI while still using many pieces of MPI infrastructure such as request objects and the progress notification routines (`MPI_Test`, `MPI_Wait`). The generalized request design as it stands, however, has deficiencies regarding typical use cases. These deficiencies are particularly evident in environments that do not support threads or signals, such as the leading petascale systems (IBM Blue Gene/L, Cray XT3 and XT4). This paper examines these shortcomings, proposes extensions to the interface to overcome them, and presents implementation results.

1 Introduction

In a message-passing environment, a nonblocking communication model often makes a great deal of sense. Implementations have flexibility in optimizing communication progress; and, should asynchronous facilities exist, computation can overlap with the nonblocking routines.

MPI provides a host of nonblocking routines for independent communication, and MPI-2 added nonblocking routines for file I/O. When callers post nonblocking MPI routines, they receive an MPI request object, from which the state of the nonblocking operation can be determined. Generalized requests, added as part of the MPI-2 standard [1], provide a way for users to define new nonblocking operations. Callers of these user-defined functions receive a familiar request object and can use the same test and wait functions as a native request object. A single interface provides a means to test communication, I/O, and user-defined nonblocking operations.

Generalized requests, unfortunately, are difficult to use in some environments. Our experience with generalized requests comes from using them to implement nonblocking I/O in the widely available ROMIO MPI-IO implementation [2].

In the absence of generalized requests, ROMIO defines its own ROMIO-specific request objects to keep track of state in its nonblocking MPI-IO routines. With these custom objects, ROMIO does not need to know the internals of a

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 223–232, 2007.

given MPI implementation. The usual MPI request processing functions, however, cannot operate on ROMIO's custom objects, so ROMIO must also export its own version of the MPI test and wait routines (`MPIO_TEST`, `MPIO_WAIT`, etc.). Moreover, these custom objects and routines are not standards-conformant.

With the implementation of MPI-2 on many more platforms now, ROMIO can use generalized requests instead of custom requests and functions. Generalized requests allow ROMIO to adhere to the MPI standard and provide fewer surprises to users of ROMIO's nonblocking routines. Unfortunately, the current definition of generalized requests makes it difficult (or in some instances impossible) to implement truly nonblocking I/O. To carry out asynchronous I/O with generalized requests, ROMIO must spawn a thread or set up a signal handler that can then test and indicate an asynchronous operation has completed. Since threads or signal handlers cannot be used in all environments, however, an alternative approach is desireable.

In this work we examine the shortcomings of the existing generalized request system, propose improvements to the generalized request design, and discuss the benefits these improvements afford.

2 MPI Requests vs. Generalized Requests

The MPI standard addresses the issue of progress for nonblocking operations in Section 3.7.4 of [3] and Section 6.7.2 of [1]. MPI implementations have some flexibility in how they interpret these two sections. The choice of a weak interpretation (progress occurs only during MPI calls) or a strict interpretation (progress can occur at any point in time) has a measurable impact on performance, particularly when the choice of progress model affects the amount of overlap between computation and communication [4].

The MPI-2 standard addresses the issue of progress for generalized requests by defining a super-strict model in which no progress can be made during an MPI call. When creating generalized requests, users must ensure all progress occurs outside of the context of MPI.

Here's how one uses generalized requests to implement a new nonblocking operation. The new operation calls `MPI_GREQUEST_START` to get an MPI request object. After the operation has finished its task, a call to `MPI_GREQUEST_COMPLETE` marks the request as done. However, the completion call will never be invoked by any MPI routine. All progress for a generalized request must be made outside of the MPI context. Typically a thread or a signal handler provides the means to make progress.

When we used generalized requests to implement nonblocking I/O routines in ROMIO, we found this super-strict progress model limiting. In many situations we do not want to or are unable to spawn a thread. Moreover, we recognized that we could effectively apply generalized requests to more situations if we could relax the progress model. We could also achieve a greater degree of overlap between computation and file I/O.

3 Asynchronous File I/O

To illustrate the difficulties using generalized requests with ROMIO, we use asynchronous file I/O as an example. The most common model today is POSIX AIO [5], but Win32 Asynchronous I/O [6] and the PVFS nonblocking I/O interfaces [7] share a common completion model with POSIX AIO.

Table 1. Typical functions for several AIO interfaces

	POSIX AIO	Win32 AIO	PVFS v2
Initiate	`aio_write`	`WriteFileEx`	`PVFS_isys_write`
Test	`aio_error`	`SleepEx`	`PVFS_sys_test`
Wait	`aio_suspend`	`WaitForSingleObjectEx`	`PVFS_sys_wait`
Wait (all)	`aio_suspend`	`WaitForMultipleObjectsEx`	`PVFS_sys_waitall`

In Table 1 we show a few of the functions found in these three AIO interfaces. The completion model looks much like that of MPI and involves two steps: post an I/O request and then, at some point, test or wait for completion of that request. After posting I/O operations, a program can perform other work while the operating system asynchronously makes progress on the I/O request. The operating system has the potential to make progress in the background though all work could occur in either the initiation or the test/wait completion call. This model lends itself well to programs with a single execution thread.

We note that POSIX AIO does define an alternative mechanism to indicate completion via real-time signals. This signal-handler method fits well only with POSIX AIO, however. Neither the other AIO interfaces nor other situations where work is occurring asynchronously can make effective use of signals, and so we will not consider them further.

4 Generalized Request Deficiencies

ROMIO, one of the earliest and most widely deployed MPI-IO implementations, has portability as a major design goal. ROMIO strives to work with any MPI implementation and on all platforms. Because of this portability requirement, ROMIO cannot always use threads. While POSIX threads are available on many platforms, they are notably not available on the Blue Gene/L or the Cray XT3 and XT4 machines, for example.

As discussed in Section 2, the requirement of a super-strict progress model for generalized requests makes it difficult to create new nonblocking operations without spawning a thread. Under this super-strict progress model, common asynchronous I/O interfaces have no good thread-free mechanism by which to invoke their completion routine.

Consider the code fragment in Figure 1 implementing `MPI_File_iwrite`. If the implementation wishes to avoid spawning a thread, it must block: there is no other way to invoke `aio_suspend` and `MPI_Grequest_complete` yet, a thread or

```
MPI_File_iwrite(..., *request) {
    struct aiocb write_cb = { ... }

    aio_write(&write_cb)
    MPI_Grequest_start(..., request)
    aio_suspend(write_cb, 1, MAX_INT)
    MPI_Grequest_complete(request)
    return;
}
```

Fig. 1. A thread-free way to use generalized requests. In the current generalized request design, the post and the test for completion of an AIO operation and the call to `MPI_GREQUEST_COMPLETE` must all happen before the routine returns. Future `MPI_Wait` routines will return immediately as the request is already completed.

signal handler is unnecessary in the file AIO case: the operating system takes care of making progress. This pseudocode is not a contrived example. It is essentially the way ROMIO must currently use generalized requests. The current generalized request design needs a way for the MPI test and wait routines to call a function that can determine completion of such AIO requests.

Other Interfaces. In addition to AIO, other interfaces might be able to make use of generalized requests were it not for portability issues. Coupled codes, such as those used in weather forecasting, need a mechanism to poll for completion of various model components. This mechanism could use generalized requests to initiate execution and test for completeness. Nonblocking collective communication lends itself well to generalized requests as well, especially on architectures with hardware-assisted collectives. These interfaces, however, must accommodate the lack of thread support on Blue Gene/L and Cray XT series machines and cannot use generalized requests in their current form if they wish to remain portable.

In this paper we suggest improvements to the generalized request interface. We use asynchronous I/O as an illustrative example. However, the benefits would apply to many situations such as those given above where the operating environment can do work on behalf of the process.

5 Improving the Generalized Request Interface

As we have shown, AIO libraries need additional function calls to determine the state of a pending operation. We can accommodate this requirement by extending the existing generalized request functions. We propose an `MPIX_GREQUEST_START` function similar to `MPI_GREQUEST_START`, but taking an additional function pointer (`poll_fn`) that allows the MPI implementation to make progress on pending generalized requests. We give the prototype for this function in Figure 3 in the Appendix.

When the MPI implementation tests or waits for completion of a generalized request, the poll function will provide a hook for the appropriate AIO

completion function. It may be helpful to illustrate how we imagine an MPI implementation might make use of this extension for the test and wait routines ({MPI_TEST,MPI_WAIT}{,ALL,ANY,SOME}). All cases begin by calling the request's user-provided poll_fn. For the wait routines, the program continues to call poll_fn until either at least one request completes (wait, waitany, waitsome) or all requests complete (wait, waitall).

An obvious defect of this approach is that the MPI_WAIT{ANY/SOME/ALL} and MPI_WAIT functions must poll (i.e., busy wait). The problem is that we do not have a single wait function that we can invoke. In Section 7 we provide a partial solution to this problem.

6 Results

We implemented MPIX_GREQUEST_START in an experimental version of MPICH2 [8] and modified ROMIO's nonblocking operations to take advantage of this extension. Without this extension, ROMIO still uses generalized requests, but does so by carrying out the blocking version of the I/O routine before the nonblocking routine returns. With the extension, ROMIO is able to initiate an asynchronous I/O operation, use generalized requests to maintain state of that operation, and count on our modified MPICH2 to invoke the completion routine for that asynchronous I/O operation during test or wait. This whole procedure can be done without any threads in ROMIO.

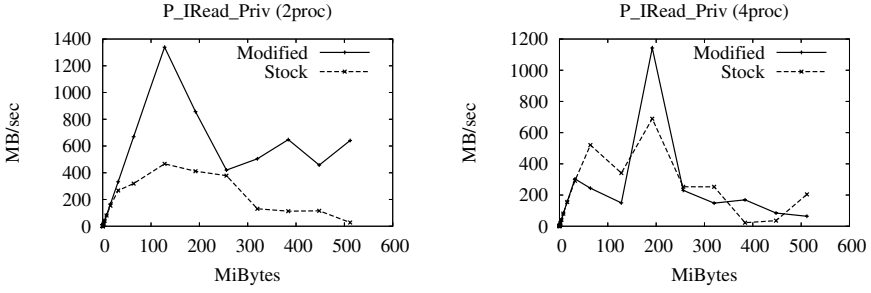
Quantifying performance of a nonblocking file operation is not straightforward. Ideally, both the I/O and some unit of work execute concurrently and with no performance degradation. Capturing both performance and this measure of "overlap" can be tricky.

Nonblocking writes introduce an additional consideration when measuring performance. Write performance has two factors: when the operating system says the write is finished, and when the write has been flushed from buffers to disk. Benchmark results for both old and new MPICH2 implementations look quite similar, since MPI_FILE_SYNC dominates the time for both implementations. We will therefore focus on performance of the more straightforward read case.

We used the Intel®MPI Benchmarks package [9]. Our results are for "optional" mode, only because we increased the maximum message size from 16 MB to 512 MB in order to see how performance varied across a wider scale of I/O sizes. Our test platform is a dual dual-core Opteron (4 cores total), writing to a local software RAID-0 device.

We depict the results of the P_IreadPriv benchmark in Figure 2(a) (2 processes) and Figure 2(b) (4 processes). This MPI-IO test measures nonblocking I/O performance when each process reads data from its own file (i.e., one file per process) while a synthetic CPU-heavy workload runs concurrently. The benchmark varies the size of the nonblocking I/O requests while keeping the CPU workload fixed (0.1 seconds).[1] When comparing two MPI implementations,

[1] This benchmark computes an "overlap" factor, but the computation in this case gave odd and inconsistent results.



(a) 2 Processes (b) 4 Processes

Fig. 2. `P_IRead_Priv` test with two MPI processes. "Stock" depicts standard (blocking) generalized requests. "Modified" shows performance with our improvements.

we found computing the effective bandwidth at a given request size yielded a useful metric for evaluating relative overlap. "Effective bandwidth" in this case means the size of a request divided by the (inclusive) time taken to post the request, run the CPU-heavy workload, and detect completion of that request. Higher effective bandwidth means a higher degree of overlap between I/O and computation.

Both graphs have three regions of interest. For small I/O sizes, true nonblocking operations give little if any benefit. As the amount of I/O increases, however, effective bandwidth increases when the MPI implementation can carry out I/O asynchronously. Asynchronous I/O benefits most — nearly three times at peak — if there are spare CPUs, but even in the fully subscribed case we see almost a doubling of peak performance. At large enough request sizes, the amount of I/O dwarfs the fixed amount of computation, limiting the opportunities for I/O and computation to overlap. Even so, asynchronous I/O on this platform appears to benefit significantly from the unused cores in the two-process case. We suspect polling in the MPI implementation might have an impact on I/O performance. In Section 7 we propose a refinement that can limit busy-waiting.

We note that the work described in this paper *enables* asynchronous I/O. Whether asynchronous I/O is beneficial or not depends on many factors, such as application workload and the quality of a system's AIO libraries. Finding the ideal balance in overlapping I/O and computation is a fascinating area of research but is beyond the scope of this paper.

7 Further Improvements: Creating a Generalized Request Class

With this simple extension to generalized requests we have already achieved our main goals: ROMIO has a hook by which it can determine the status of a pending AIO routine, and can do so without spawning a thread. If we observe

that generalized requests are created with a specific task in mind, we can further refine this design.

In the AIO case, all callers are going to use the same test and wait routines. In POSIX AIO, for example, a nonblocking test for completion of an I/O operation (read or write) can be carried out with a call to `aio_error`, looking for `EINPROGRESS`. AIO libraries commonly provide routines to test for completion of multiple AIO operations. The libraries also have a routine to block until completion of an operation, corresponding to the `MPI_WAIT` family.

We can give implementations more room for optimization if we introduce the concept of a generalized request class. `MPIX_GREQUEST_CLASS_CREATE` would take the generalized request query, free, and cancel function pointers already defined in MPI-2 along with our proposed poll function pointer. The routine would also take a new "wait" function pointer. Initiating a generalized request then reduces to instantiating a request of the specified class via a call to `MPIX_GREQUEST_CLASS_ALLOCATE`. Prototypes for these routines are given in Figure 4 in the Appendix.

At first glance this may appear to be just syntax: why all this effort just to save passing two pointers? One answer is that in ROMIO's use of generalized requests, the query, free, and cancel methods are reused multiple times; Hence, a generalized request class would slightly reduce duplicated code.

A more compelling answer lies in examining how to deal with polling. By creating a class of generalized requests, we give implementations a chance to optimize the polling strategy and minimize the amount of time consuming CPU while waiting for request completion.

Refer back to Figure 2(b), where the unmodified, blocking MPICH2 outperforms the modified MPICH2 at the largest I/O request size. At this point, I/O takes much longer to compute than the computation. All available CPUs are executing the benchmark and polling repeatedly inside `MPI_Waitall` until the I/O completes. The high CPU utilization, aside from doing no useful work, also appears to be interfering with the I/O transfer.

Our proposed generalized request class adds two features that together solve the problem of needlessly consuming CPU in a tight testing loop. First, we introduce `wait_fn`, a hook for a blocking function that can wait until completion of one or more requests. If multiple generalized requests are outstanding, an implementation cannot simply call a user-provided wait routine (created with a specific generalized request in mind) on all of them. If all the outstanding requests are of the same generalized request class, however, the implementation might be able to pass several or even all requests to a user-provided wait routine, which in turn could complete multiple nonblocking operations. By avoiding repeated polling and aggregating multiple requests, our generalized request class thus can make processing user-defined nonblocking operations more efficient, particularly in those MPI functions such as `MPI_WAITALL` that take multiple requests.

Generalized request classes also open the door for the MPI implementation to learn more about the behavior of these user-provided operations, and potentially adapt. We imagine that an MPI implementation could keep timing information

or other statistics about a class of operations and adjust timeouts or otherwise inform decisions in the same way Worringen automatically adjusts MPI-IO hints in [10]. Implementations cannot collect such statistics without a request class, since those implementations can glean meaningful information only by looking at generalized requests implementing a specific feature.

8 Conclusions

Generalized requests in their current form do much to simplify the process of creating user-provided nonblocking operations. By tying into an implementation's request infrastructure, users avoid reimplementing request bookkeeping. Unfortunately, while generalized requests look in many ways like first-class MPI request objects, the super-strict progress model hinders their usefulness. Whereas an MPI implementation is free to make progress for a nonblocking operation in the test or wait routine, generalized requests are unable to make progress in this way. This deficiency manifests itself most when interacting with common asynchronous I/O models, but it is also an issue when offloading other system resources.

We have presented a basic extension to the generalized request design as well as a more sophisticated class-based design. In reviewing the MPI Forum's mailing list discussions about generalized requests, we found early proposals advocating an approach similar to ours. A decade of implementation experience and the maturity of AIO libraries show that these early proposals had merit that perhaps went unrecognized at the time. For example, at that time it was thought that using threads would solve the progress problem, but today we are faced with machines for which threads are not an option.

Our extensions would greatly simplify the implementation of nonblocking I/O operations in ROMIO or any other library trying to extend MPI with custom nonblocking operations. Class-based approaches to making progress on operations would alleviate some of the performance concerns of using generalized requests.

Unlike many MPI-2 features, generalized requests have seen neither widespread adoption nor much research interest. We feel the extensions proposed in this paper would make generalized requests more attractive for library writers and for those attempting to use MPI for system software, in addition to opening the door for additional research efforts.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

1. The MPI Forum: MPI-2: Extensions to the message-passing interface. The MPI Forum (July 1997)
2. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems, pp. 23–32 (1999)
3. Message Passing Interface Forum: MPI: A message-passing interface standard. Technical report (1994)
4. Brightwell, R., Riesen, R., Underwood, K.: Analyzing the impact of overlap, offload, and independent progress for MPI. The International Journal of High-Performance Computing Applications 19(2), 103–117 (2005)
5. IEEE/ANSI Std. 1003.1: Single UNIX specification, version 3 (2004 edition)
6. Microsoft corporation: Microsoft Developer Network Online Documentation (accessed 2007), http://msdn.microsoft.com
7. PVFS development team: The PVFS parallel file system (accessed 2007), http://www.pvfs.org/
8. MPICH2 development team: MPICH2. http://www.mcs.anl.gov/mpi/mpich2
9. Intel GmbH: Intel MPI benchmarks. http://www.intel.com
10. Worringen, J.: Self-adaptive hints for collective I/O. In: Proceedings of the 13th European PVM/MPI User's Group, Bonn, Germany, pp. 202–211 (2006)

Appendix: Function Prototypes

In this paper we have proposed several new MPI routines. Figure 3 and Figure 4 give the C prototypes for these routines.

```
int MPIX_Grequest_start(
        MPI_Grequest_query_function *query_fn,
        MPI_Grequest_free_function *free_fn,
        MPI_Grequest_cancel_function *cancel_fn,
        MPIX_Grequest_poll_function *poll_fn,
        void *extra_state,
        MPI_Request *request)
typedef int MPIX_Grequest_poll_fn(
        void *extra_state,
        MPI_Status *status);
```

Fig. 3. Prototypes for generalized request poll extension

```
typedef int MPIX_Grequest_wait_fn(
        int count,
        void *array_of_states,
        double timeout,
        MPI_Status *status);
int MPIX_Grequest_class_create(
        MPI_Grequest_query_function *query_fn,
        MPI_Grequest_free_function *free_fn,
        MPI_Grequest_cancel_function,
        MPIX_Grequest_poll_fn,
        MPIX_Grequest_wait_fn,
        MPIX_Request_class *greq_class);
int MPIX_Grequest_class_allocate(
        MPIX_Request_class greq_class,
        void *extra_state
        MPI_Request *request)
```

Fig. 4. Prototypes for generalized request classes and blocking wait function

Transparent Log-Based Data Storage in MPI-IO Applications

Dries Kimpe[1,2], Rob Ross[3], Stefan Vandewalle[1], and Stefaan Poedts[2]

[1] Technisch-Wetenschappelijk Rekenen, K.U.Leuven, Celestijnenlaan 200A, 3001 Leuven, België
Dries.Kimpe@cs.kuleuven.be
[2] Centrum voor Plasma-Astrofysica, K.U.Leuven, Celestijnenlaan 200B, 3001 Leuven, België
[3] Argonne National Laboratory, 9700 S Cass Ave, 60439 Argonne, IL

Abstract. The MPI-IO interface is a critical component in I/O software stacks for high-performance computing, and many successful optimizations have been incorporated into implementations to help provide high performance I/O for a variety of access patterns. However, in spite of these optimizations, there is still a large performance gap between "easy" access patterns and more difficult ones, particularly when applications are unable to describe I/O using collective calls.

In this paper we present LogFS, a component that implements log-based storage for applications using the MPI-IO interface. We first discuss how this approach allows us to exploit the temporal freedom present in the MPI-IO consistency semantics, allowing optimization of a variety of access patterns that are not well-served by existing approaches. We then describe how this component is integrated into the ROMIO MPI-IO implementation as a stackable layer, allowing LogFS to be used on any file system supported by ROMIO. Finally we show performance results comparing the LogFS approach to current practice using a variety of benchmarks.

1 Introduction

Dealing with complex I/O patterns remains a challenging task. Despite all optimizations, there is still a huge difference in I/O performance between simple (contiguous) and complex (non-contiguous) access patterns [2] This difference can be attributed to physical factors (non-contiguous patterns typically cause read-modify-write sequences and require time-consuming seek operations) and to software issues (complex patterns require more memory and processing). Recent papers concentrated on reducing the software overhead associated with processing complex I/O patterns [4,5,6].

In computational science *defensive I/O* is common: applications write checkpoints in order to provide a way to rollback in the event that a system failure terminates the application prematurely. In the event that an application

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 233–241, 2007.

successfully executes for an extended period, these checkpoints may never actually be read. So while the application might exhibit complex access patterns during checkpoint write, if we can defer the processing of the complex pattern, we might attain much higher throughputs than otherwise possible. Of course, some mechanism must be available for post-processing the complex pattern in the event that the data is read, ideally with little or no additional overhead. If this post-processing mechanism is fast enough, this approach could be used as a general-purpose solution as well.

The MPI-IO interface is becoming the standard mechanism for computational science applications to interact with storage, either directly or indirectly via high-level libraries such as HDF5 and Parallel netCDF [9]. This makes the MPI-IO implementation an ideal place to put I/O optimizations. The MPI-IO default consistency semantics are more relaxed than the traditional POSIX semantics that most file systems strive to implement. In particular, the MPI-IO default semantics specify that only local changes are visible to an application process until an explicit file synchronization call is made. This aspect of the standard will enable us to further optimize our implementation.

In this paper we present LogFS, an extension to the ROMIO MPI-IO implementation [1] designed to provide log-based storage of file data in parallel applications. The functionality is provided transparently to the user, it follows the MPI-IO consistency semantics, and a mechanism is provided for reconstituting the canonical file so that UNIX applications can subsequently access the file. In Section 2 we describe the LogFS approach. In Section 3 we show how this approach has significant performance benefits in write-heavy workloads. In Section 4 we conclude and discuss future directions for this work.

1.1 Related Work

Two groups are actively pursuing research in MPI-IO optimizations that are relevant to this work. The group at Northwestern University has been investigating mechanisms for incorporating cooperative caching into MPI-IO implementations to provide what is effectively a large, shared cache. They explore using this cache for both write-behind, to help aggregate operations, and to increase hits in read-heavy workloads [12,13]. Our work is distinct in that we allow our "cache" to spill into explicit log files, and we perform this caching in terms of whole accesses rather than individual pages or blocks. Their implementations to date have also relied on the availability of threads, MPI-2 passive-target one-sided communication, or the Portals interface, limiting applicability until passive-target operations become more prevalent on large systems. Our work does not have these requirements.

Yu et. al. have been investigating mechanisms for improving parallel I/O performance with the Lustre file system. Lustre includes a feature for joining previously created files together into a new file. They leverage these features to create files with more efficient striping patterns, leading to improved performance [11]. Their observations on ideal stripe widths could be used to tune stripe widths for our log files or to improve the ration of creation of the final canonical file on systems using Lustre.

2 LogFS

The LogFS extension to ROMIO provides independent, per-process write logging for applications accessing files via MPI-IO. On each process, logging is separated into a log file containing the data to be written, called the *datalog* and a log file holding metadata about these writes such as location and epoch, called the *metalog*. A global *logfs file* is stored while the file is in log format and maintains a list of the datalogs and metalogs. Together these files maintain sufficient data that the correct contents of the file can be generated at any synchronization point.

The logfs file is initially created using `MPI_MODE_EXCL` and `MPI_MODE_CREATE`. Note that in a production implementation we would need to manage access to this file so that the file was open by only one MPI application if logging was in progress. This could be managed using an additional global lockfile.

The datalog and metalog files are opened independently with `MPI_COMM_SELF` and are written sequentially in contiguous blocks, regardless of the application's write pattern, hiding any complexity in the write pattern and deferring the transformation of the data into the canonical file organization (i.e. the traditional POSIX file organization) until a later time.

The data logfile contains everything written to the file by the corresponding process. Data is written in large contiguous blocks corresponding to one or more MPI-IO write operations. This lends itself to high performance on most parallel file systems, because there is no potential for write lock contention.

The metadata logfile records, for every write operation by the process, the file offset (both in the real file and in the datalog) and the transfer size. In addition to write operations, the metadata log also tracks `MPI_File_sync`, `MPI_File_set_size` and `MPI_File_set_view`. However, this is done in a lazy fashion: only changes actually needed to accurately replay the changes are stored. Calling sequences without effect to the final file, such as repeatedly changing the file view without actually writing to the file, are not recorded in the metalog. Of these operations, all have a fixed overhead, except for `MPI_File_set_view`. Currently, datatypes are stored as lists of $\langle$*offset, size*$\rangle$ pairs.

It was observed in early parallel I/O studies that parallel applications often perform many, small, independent I/O operations [10]. This type of behavior continues today, and in some cases high-level I/O libraries can contribute through metadata updates performed during I/O. Keeping this in mind, LogFS can additionally use a portion of local memory for aggregation of log entries. This aggregation allows LogFS to more efficiently manage logging of I/O operations and convert many small I/O operations into a fewer number of larger contiguous ones, again sequential in file.

2.1 Creating the Canonical File for Reading

By default, LogFS tries to postpone updating the canonical file for as long as possible. In some situations, such as files opened in write-only mode, even closing the file will not necessarily force a replay. This allows for extremely efficient checkpointing.

When a LogFS file is opened in `MPI_MODE_RDONLY`, the canonical file is automatically generated if logs are still present. The canonical file is generated through a collective "replay" of the logs. In our implementation we assume that the number of replay processes is the same as the original number of writer processes, but replay processes could manage more than one log at once.

Replay occurs in *epochs* corresponding to writes that occur between synchronization points. By committing all writes from one epoch before beginning the next, we are able to correctly maintain MPI-IO consistency semantics without tracking the timing of individual writes. Each process creates an in memory rtree[8], a spatial data structure allowing efficient range queries. Replayers move through the metalog, updating their rtree with the location of the written data in the datalog. When the replayer hits the end of an epoch or the processed data reaches a certain configurable size, the replayer commits these changes to the canonical file. To accomplish a commit, the replayer process reads the data from the datalog into a local buffer, calls `MPI_File_set_view` to define the region(s) in the canonical file to modify, then calls `MPI_File_write_all` to modify the region(s). Processes use `MPI_Allreduce` between commits to allow all processes to complete one epoch before beginning the next. This approach to replay always results in large, collective I/O operations, allowing any underlying MPI-IO optimizations to be used to best effect [7].

This also enables the user to only replay those files that are actually needed. With this system, LogFS is transparent to all applications using MPI-IO; replay will happen automatically when needed. However, it is often the case that post-processing tools are not written using MPI-IO. In the climate community, for example, Parallel netCDF is often used to write datasets in parallel using MPI-IO, but many post-processing tools use the serial netCDF library, which is written to use POSIX I/O calls. For those situations, a small stand-alone utility is provided that can force the replay of a LogFS file.

An additional MPI-IO hint, `replay-mode` is understood by the LogFS-enabled ROMIO. When this is set to "replay-on-close", replay is automatically performed when a LogFS file is closed after writing. The stand-alone tool simply opens the LogFS file for writing with this hint set, then closes the file. Note that with this approach as many processes as originally wrote the LogFS file may be used to replay in parallel.

2.2 Mixed Read and Write Access

The LogFS system is obviously designed for situations where writes and reads are not mixed. However, for generality we have implemented two mechanisms for supporting mixed read and write workloads under LogFS.

When `MPI_MODE_RDRW` is selected for a file, MPI consistency semantics require that a process is always able to read back the data it wrote; Unless atomic mode is also enabled, data written by other processes only has to become visible after `MPI_File_sync` is called (or after closing and re-opening the file, which performs an implicit sync). If a user chooses both atomic mode and `MPI_MODE_RDRW`, LogFS

optimizations are not appropriate, and we will ignore that case in the remainder of this work.

In order to guarantee that data from other processes is visible at read time, we replay local logs on each process at synchronization points. Replay consists of local independent reads of logs followed by collective writes of this data in large blocks. This has the side-effect of converting all types of application access (independent or collective, contiguous or noncontiguous) into collective accesses, increasing performance accordingly [7].

We have two options for guaranteeing that data written locally is returned on read operations prior to synchronization. A simple option is for processes to ensure that the canonical file is up-to-date on read; this may be accomplished by performing a local replay in the event of a read operation. In this case, only the first read operation will be slow, all subsequent reads will continue at native speeds. However, this method performs badly with strongly mixed I/O sequences; Frequent reads force frequent log replays, and the efficiency of write aggregation diminishes with increased replay frequencies.

Another option is for each process to track regions written locally since the last sync operation. If those regions overlap with parts of a read request, data needs to be read from the datalog. Accesses to unchanged regions may be serviced using data from canonical file. If a very large number of writes occur, and the memory cost of tracking each individual region becomes too high, we have the option of falling back to our first option and completely replaying the local log, removing the need to track past regions.

To efficiently track write regions during `MPI_MODE_RDRW` mode, every process maintains an in-memory rtree at run-time. For every written region, the rtree records the location of that data in the datalog of the process, and on every write operation this rtree is updated, in a manner similar to the approach used in replay.

The rtree then provides us with an efficient mechanism for determining if local changes have been made since the last synchronization, and if so, where that data is located in the datalog. With this scheme a read request gets transformed by LogFS into at most two read operations; one to read data from the datalog, and one to read any remaining data from the canonical file.

Unfortunately, tracking all affected regions in long-living files with lots of fragmented write accesses can lead to large rtree descriptions. In these cases we are forced to either update the canonical file with local changes, to shrink the rtree, or stop tracking writes all together and fall back to our original option.

2.3 Implementing in ROMIO

LogFS is implemented as a component integrated into ROMIO[1]. ROMIO incorporates an interface for supporting multiple underlying file systems called ADIO. We prototyped two approaches for implementing LogFS.

LogFS ADIO Implementation. The first approach was to implement LogFS as a new ADIO component. In this approach LogFS appears as a new file system

type, but internally it makes use of some other ADIO implementation for performing file I/O. For example, a user opening a new file on a PVFS file system using the "logfs:" prefix on their file name would create a new LogFS-style file with logs and canonical file stored on the PVFS file system. A consequence of this approach is that LogFS can be enabled on any filesystem supported by ROMIO, and the file may be written in the usual "normal" data representation.

Under a strict interpretation of the MPI-IO standard, changes to a file in the "normal" (or "external32") data representation must be made visible to other applications at synchronization points, unless the file is opened with `MPI_MODE_UNIQUE_OPEN`. To meet this strict interpretation of the standard, LogFS must perform a full replay at synchronization points if unique open is not specified, even when in write-only mode. This approach is only most effective when applications use the unique open mode.

LogFS as a Data Representation. Our second approach was to implement LogFS as ROMIO's "internal" data representation, a somewhat creative interpretation of the internal data representation specification. To function as a data representation, LogFS must intercept all file access operations. For this purpose, a layering technique for ADIO components was developed which allowed transparent interception of all ADIO methods.

When the user changes to our internal data representation (using `MPI_File_set_view`, the LogFS ADIO is layered on top of the active ADIO driver for the file. One difference between this approach and the ADIO approach is that the data representation may be changed through `MPI_File_set_view` at any time, so if the view is later restored to its original setting, the logfiles are immediately replayed and the canonical file created.

According to the standard, the format of a file stored in the internal data representation is not known. This means that we can force applications to open the file and change the data representation back to "native" prior to access by application not using MPI-IO. This hook allows us to avoid the need to replay logs at synchronization points in the general case.

3 Performance Results

In this section we show results of experiments comparing the LogFS approach to a stock ROMIO implementation (included in mpich 1.0.5p4). As the base filesystem, PVFS[14] version 2.6.3 was used. The filesystem was configured to use TCP (over gigabit ethernet) as network protocol, with 4 I/O servers and 1 metadataserver. For testing the filesystem, 16 additional nodes were used.

Before we present our results we will first quantitatively describe the overhead incurred by our logging process.

3.1 Overhead

There is a certain amount of overhead introduced by first recording I/O operations in the logfiles. During the write-phase, overhead consists of meta data

(describing the I/O operation) and actual data (the data to be written), both stored in the logfiles. Likewise, during replay, everything read from the logfiles can be considered overhead. Table 1 indicates per-operation overhead based on the storage format described in 2.

Table 1. LogFS logfile overhead

Operation	Log File Overhead	
	Metalog	Datalog
`MPI_File_write`	datalog offset, write offset (2x `MPI_OFFSET`)	*datatype size* × *count*
`MPI_File_set_view`	displacement(`MPI_OFFSET`) + etype(*flatlist*) + filetype(*flatlist*)	0
`MPI_File_sync`	epoch number(`MPI_INT`)	0
`MPI_File_set_size`	filesize(`MPI_OFFSET`)	0

Many scientific applications perform regular checkpointing. Typically, only the latest or a small number of checkpoints are kept; In this case, most of the data written to the checkpoint file will never survive; it will be overwritten during run-time or deleted shortly after the application terminates. Although all data will be forced out to the datalog, in the event that data is overwritten it will never be read again because a replay of the metalog will only keep track of the most recent data.

In the worst case when data is not overwritten, all data will be read again during replay. However, since data in the metalog is accessed sequentially and in large blocks, these transfers typically reach almost full filesystem bandwidth.

3.2 Results

For testing, we choose the well known "noncontig" benchmark. Noncontig partitions the test file in vectors of a fixed size, allocating them in a round-robin fashion to every CPU. This generates a non-contiguous regular strided access pattern. Figure 1 shows how the LogFS write bandwidth compares to that of a stock ROMIO implementation.

The results clearly show how LogFS is capable of transforming extremely inefficient access patterns (such as the independent non-contiguous pattern in the left figure) into faster ones. Using LogFS, a peak write bandwidth of approx. 300 MB/s is reached. Without LogFS, peak bandwidth is only approx. 20 MB/s for collective access, and only around 2 MB/s for independent access.

To see how closely LogFS gets to the maximum write bandwidth possible, a small test program ("contwrite") was created that directly writes large blocks – the same size of the write-combining buffers used in two-phase and LogFS – to the filesystem. The results can be seen in Figure 1.

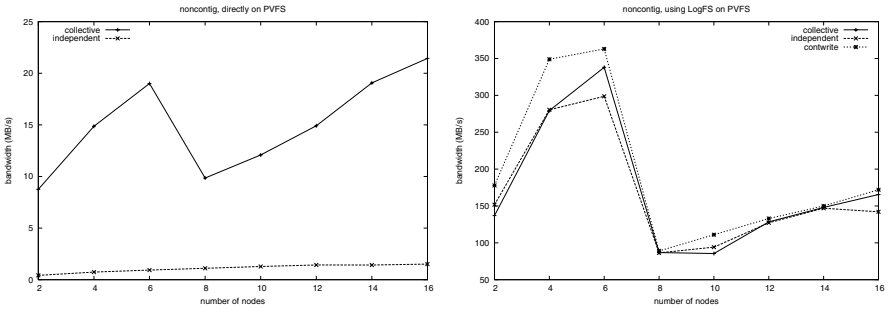


Fig. 1. noncontig benchmark (vector count = 8388608)

4 Conclusions and Future Work

LogFS is capable of enhancing write performance of programs using complex file access patterns. Its layered design and modest file system requirements allow the approach to be employed on a wide variety of underlying file systems. For application checkpoints that might never be read again, performance can be improved by at least a factor of 10.

LogFS is primarily targeted at write-only (or write-heavy) I/O workloads. However, much of the infrastructure implemented for LogFS may be reused to implement independent per-process caching in MPI-IO. We are actively pursuing this development. This approach provides write aggregation benefits, can transform a larger fraction of I/O operations into collective ones, and has benefits for read-only and read-write workloads. Similar to LogFS, per-process caching doesn't require MPI threads or passive-target one-sided MPI operations, meaning that it can be implemented on systems such as the IBM Blue Gene/L [3] and Cray XT3 that lack these features.

Currently, LogFS creates one logfile for every process opening the file. When running on large numbers of processes, this leads to a huge amount of logfiles. To avoid this, we are considering sharing logfiles between multiple processes. This approach is complicated by the need for processes opening files in read/write mode to "see" local changes between synchronization points, because this means that multiple processes might need to read the same log files at runtime.

References

1. Thakur, R., Gropp, W., Lusk, E.: An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation, pp. 180–187 (1996)
2. Kimpe, D., Vandewalle, S., Poedts, S.: On the Usability of High-Level Parallel IO in Unstructured Grid Simulations. In: Proceedings of the 13th EuroPVM/MPI Conference, pp. 400–401 (2007)

3. Allsopp, N., Follows, J., Hennecke, M., Ishibashi, F., Paolini, M., Quintero, D., Tabary, A., Reddy, H., Sosa, C., Prakash, S., Lascu, O.: Unfolding the IBM Eserver Blue Gene Solution. International Business Machines Corporation (September 2005)
4. Worringen, J., Traff, J., Ritzdorf, H.: Improving Generic Non-Contiguous File Access for MPI-IO. In: Proceedings of the 10th EuroPVM/MPI Conference (2003)
5. Ross, R., Miller, N., Gropp, W.: Implementing Fast and Reusable Datatype Processing. In: Proceedings of the 10th EuroPVM/MPI Conference (2003)
6. Hastings, A., Choudhary, A.: Exploiting Shared Memory to Improve Parallel I/O Performance. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, Springer, Heidelberg (2006)
7. Thakur, R., Gropp, W., Lusk, E.: A case for using MPI's derived datatypes to improve I/O performance. In: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, San Jose, CA (1998)
8. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), ACM, New York (1984)
9. Li, J., Liao, W., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., Zingale, M.: Parallel netCDF: A High-Performance Scientific I/O Interface. In: Proceedings of SC2003 (2003)
10. Purakayastha, A., Ellis, C., Kotz, D., Nieuwejaar, N., Best, M.: Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor. In: Proceedings of the Ninth International Parallel Processing Symposium (1995)
11. Yu, W., Vetter, J., Canon, R., Jiang, S.: Exploiting Lustre File Joining for Effective Collective IO. In: Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), IEEE Computer Society Press, Los Alamitos (2007)
12. Coloma, K., Choudhary, A., Liao, W., Ward, L., Tideman, S.: DACche: Direct Access Cache System for Parallel I/O. In: the 2005 Proceedings of the International Supercomputer Conference (2005)
13. Liao, W., Ching, A., Coloma, K., Choudhary, A., Kandemir, M.: Improving MPI Independent Write Performance Using A Two-Stage Write-Behind Buffering Method. In: the Proceedings of the Next Generation Software (NGS) Workshop, held in conjunction with the 21th International Parallel and Distributed Processing Symposium (IPDPS), Long Beach, California (2007)
14. Carns, P.H., Ligon, W.B., Ross, III.R.B., Thakur, R.: PVFS: A Parallel File System For Linux Clusters. In: the Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, pp. 317–327 (2000)

Analysis of Implementation Options for MPI-2 One-Sided

Brian W. Barrett[1], Galen M. Shipman[1], and Andrew Lumsdaine[2]

[1] Los Alamos National Laboratory*, Los Alamos, NM 87545, USA
{bbarrett,gshipman}@lanl.gov
[2] Indiana University**, Bloomington, IN 47405, USA
lums@osl.iu.edu

Abstract. The Message Passing Interface provides an interface for one-sided communication as part of the MPI-2 standard. The semantics specified by MPI-2 allow for a number of different implementation avenues, each with different performance characteristics. Within the context of Open MPI, a freely available high performance MPI implementation, we analyze a number of implementation possibilities, including layering over MPI-1 send/receive and true remote memory access.

1 Introduction

The Message Passing Interface [1,2,3,4] (MPI) has been adopted by the high performance computing community as the communication library of choice for distributed memory systems. The original MPI specification provides for point-to-point and collective communication, as well as environment management functionality. The MPI-2 specification added dynamic process creation, parallel I/O, and one-sided communication.

The MPI-2 one-sided specification allows for implementation over send/receive or remote memory access (RMA) networks. Although this design feature has been the source of criticism [5], it also ensures maximum portability, a goal of MPI. The MPI-2 one-sided interface utilizes the concept of exposure and access epochs to define when communication can be initiated and when it must be completed. Explicit synchronization calls are used to initiate both epochs, a feature which presents a number of implementation options, even when networks support true RMA operations. This paper examines implementation options for the one-sided interface within the context of Open MPI.

* Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. LA-UR-07-3197.

** This work was supported by a grant from the Lilly Endowment and National Science Foundation grants EIA-0202048 and ANI-0330620.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 242–250, 2007.

2 Related Work

A number of MPI implementations provide support for the one-sided interface. LAM/MPI [6] provides an implementation layered over point-to-point, although it does not support passive synchronization and performance generally does not compare well with other MPI implementations. Sun MPI [7] provides a high performance implementation, although it requires all processes be on the same machine and the use of MPI_ALLOC_MEM for optimal performance. The NEC SX-5 MPI implementation includes an optimized implementation utilizing the global shared memory available on the platform [8]. The SCI-MPICH implementation provides one-sided support using hardware reads and writes [9].

MPICH2 [10] includes a one-sided implementation implemented over point-to-point and collective communication. Lock/unlock is supported, although the passive side must enter the library to make progress. The synchronization primitives in MPICH2 are significantly optimized compared to previous MPI implementations [11]. MVAPICH2 [12] extends the MPICH2 one-sided implementation to utilize InfiniBand's RMA support. MPI_PUT and MPI_GET communication calls translate into InfiniBand put and get operations for contiguous datatypes. MVAPICH2 has also examined using native InfiniBand for Lock/Unlock synchronization [13].

3 Open MPI Architecture

Open MPI [14] is a complete, open-source MPI implementation. The project is developed as a collaboration between a number of academic, government, and commercial institutions, including Indiana University, University of Tennessee, Knoxville, University of Houston, Los Alamos National Laboratory, Cisco Systems, and Sun Microsystems. Open MPI is designed to be scalable, fault tolerant, and high performance, while at the same time being portable to a variety of networks and operating systems. Open MPI utilizes a low-overhead component architecture—the Modular Component Architecture (MCA)—to provide abstractions for portability and adapting to differing application demands. In addition to providing a mechanism for portability, the MCA allows developers to experiment with different implementation ideas, while minimizing development overhead.

MPI communication is layered on a number of component frameworks, as shown in Fig. 1. The PML and OSC frameworks provide MPI send/receive and one-sided semantics, respectively. The BML, or BTL Management Layer, allows the use of multiple networks between two processes and the use of multiple upper-layer protocols on a given network, by maintainin available routes to every peer and handling scheduling across available routes. The BTL framework provides communication between two endpoints; an endpoint is usually a communication device connecting two processes, such as an Ethernet address or an InfiniBand port. The current BML/BTL implementation allows multiple upper-layer protocols to simultaneously utilize multiple communication paths. The design and implementation of the communication layer is described in detail in [15,16].

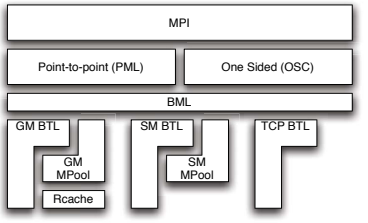


Fig. 1. Component structure for communication in Open MPI

BTL components provide two communication modes: an active-message style send/receive protocol and a remote memory access (RMA) put/get protocol. All sends are non-blocking, with a callback on local send completion, generally from a BTL-provided buffer. Receives are all into BTL-provided buffers, with a callback on message arrival. RMA operations provide callbacks on completion on the origin process and no completion callbacks on the target process (as all networks do not support remote completion events for RMA operations). All buffers used on both the origin and target must be "prepared" for use by calls to the BTL by higher-level components.

4 Open MPI One-Sided Implementation

Open MPI provides two implementations of the one-sided (OSC) framework: pt2pt and rdma. The pt2pt component is implemented entirely over the point-to-point and collective MPI functions. The original one-sided implementation in Open MPI, it is now primarily used when a network library does not expose RMA capabilities, such as Myrinet MX [17]. The rdma component is implemented directly over the BML/BTL interface and supports a variety of protocols, including active-message send/receive and true RMA. More detail on the implemnetation is provided in Section 4.2. Both components share the same synchronization implementation, although the rdma component starts communication before the synchronization call to end an epoch, while the pt2pt component does not.

4.1 Synchronization

Synchronization for both components is similar to the design used by MPICH2 [11]. Control messages are sent either over the point-to-point engine (pt2pt) or the BTL (rdma). A brief overview of the synchronization implementation follows:

Fence. MPI_WIN_FENCE is implemented with a reduce-scatter to share the number of incoming communication operations, then each process waits until the specified number of operations has completed. Communication may be started at any time during the exposure/access epoch.

General Active Target. A call to MPI_POST results in a post control message sent to every involved process. Communication may be started as soon as a post message is received from all involved processes. During MPI_COMPLETE,

all RMA operations are completed, then a control message with the number of incoming requests is sent to all peer processes. MPI_WAIT blocks until all peers send a complete message and incoming operations are completed.

Passive. Lock/Unlock synchronization does not wait for a lock to be acquired before returning from MPI_WIN_LOCK, but may start all communication as soon as a lock acknowledgment is received. During MPI_WIN_UNLOCK, a control message with number of incoming messages is sent to the peer. The peer waits for all incoming messages before releasing the lock and potentially giving it to another peer. Like other single threaded MPI implementations, our implementation currently requires the target process enter the MPI library for progress to be made.

4.2 Communication

Three communication protocols are implemented for the rdma one-sided component. For networks that support RMA operations, all three protocols are available at run-time, and the selection of protocol is made per-message.

send/recv. All communication is done using the send/receive interface of the BTL, with data copied at both sides for short messages. Long messages are transferred using the protocols provided by the PML (which may include RMA operations). Messages are queued until the end of the synchronization phase.

buffered. All communication is done using the send/receive interface of the BTL, with data copied at both sides for short messages. Long messages are transferred using the transfer protocols provided by the PML. Short messages are coalesced into the BTL's maximum eager send size. Messages are started as soon as the synchronization phase allows.

RMA. All communication for contiguous data is done using the RMA interface of the BTL. All other data is transferred using the *buffered* protocol. MPI_ACCUMULATE also falls back to the *buffered* protocol.

Due to the lack of remote completion notification for RMA operations, care must be taken to ensure that an epoch is not completed before all data transfers have been completed. Because ordering semantics of RMA operations (especially compared to send/receive operations) tends to vary widely between network interfaces, the only ordering assumed by the rdma component is that a message sent after local completion of an RMA operation will result in remote completion of the send after the full RMA message has arrived. Therefore, any completion messages sent during synchronization may only be sent after all RMA operations to a given peer have completed. This is a limitation in performance for some networks, but adds to the overall portability of the system.

5 Performance Evaluation

Latency and bandwidth micro-benchmark results are provided using the Ohio State benchmark suite. Unlike the point-to-point interface, the MPI community

has not developed a set of standard "real world" benchmarks for one-sided communication. Following previous work [11], a nearest-neighbor ghost cell update benchmark is utilized — the fence version is shown in Fig. 5. The test was also extended to call MPI_PUT once per integer in the buffer, rather than once per buffer.

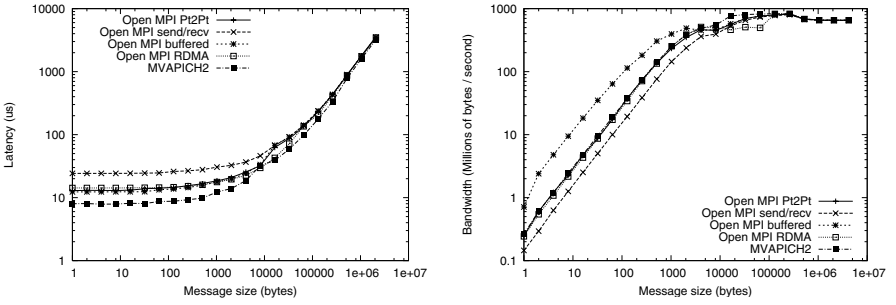
```
for (i = 0 ; i < ntimes ; i++) {
  MPI_Win_fence(MPI_MODE_NOPRECEEDE, win);
  for (j = 0 ; j < num_nbrs ; j++) {
    MPI_Put(send_buf + j * bufsize, bufsize, MPI_DOUBLE, nbrs[j],
            j, bufsize, MPI_DOUBLE, win);
  }
  MPI_Win_fence(0, win);
}
```

Fig. 2. Ghost cell update using MPI_FENCE

All tests were run on the Indiana University Department of Computer Science `Odin` cluster, a 128 node cluster of dual-core dual-socket 2.0 GHz Opteron machines, each with 4 GB of memory. Each node contains a single Mellanox InfiniHost PCI-X SDR HCA, connected to a 148 port switch. MVAPICH2 0.9.8 results are provided as a baseline. No configuration or run-time performance options were specified for MVAPICH2. The *mpi_leave_pinned* option, which tells Open MPI to leave memory registered with the network until the buffer is freed by the user rather than when communication completes, was specified to Open MPI (this functionality is the default in MVAPICH). Results are provided for the pt2pt component and all three protocols of the rdma component.

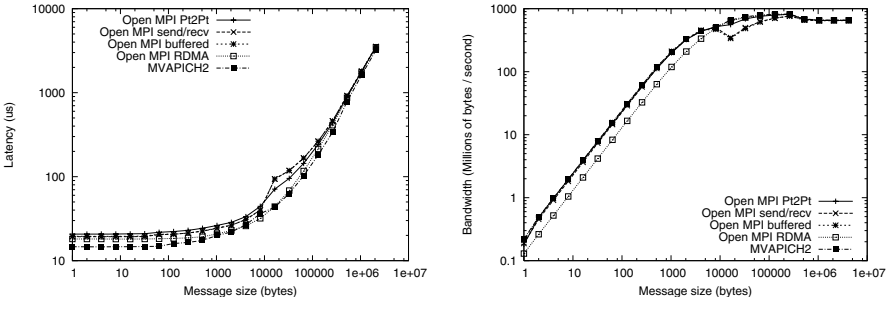
Latency/Bandwidth. Fig. 3 presents the latency and bandwidth of MPI_PUT using the Ohio State benchmarks [18]. The *buffered* protocol presents the best latency for Open MPI. Although the message coalescing of the *buffered* protocol does not improve performance of the latency test, due to only one message pending during an epoch, the protocol outperforms the *send/recv* protocol due to starting messages eagerly, as soon as all post messages are received. The *buffered* protocol provides lower latency than the *rdma* protocol for short messages because of the requirement for portable completion semantics, described in the previous section. No completion ordering is required for the *buffered* protocol, so MPI_WIN_COMPLETE does not wait for local completion of the data transfer before sending the completion count message. On the other hand, the *rdma* protocol must wait for local completion of the event before sending the completion count control message, otherwise the control message could overtake the RDMA transfer, resulting in erroneous results.

The bandwidth benchmark shows the advantage of the *buffered* protocol, as the benchmark starts many messages in each synchronization phase. The *buffered* protocol is therefore able to outperform both the *rdma* protocol and MVAPICH.



(a) Latency (b) Bandwidth

Fig. 3. Latency and Bandwidth of MPI_PUT calls between two peers using generalized active synchronization

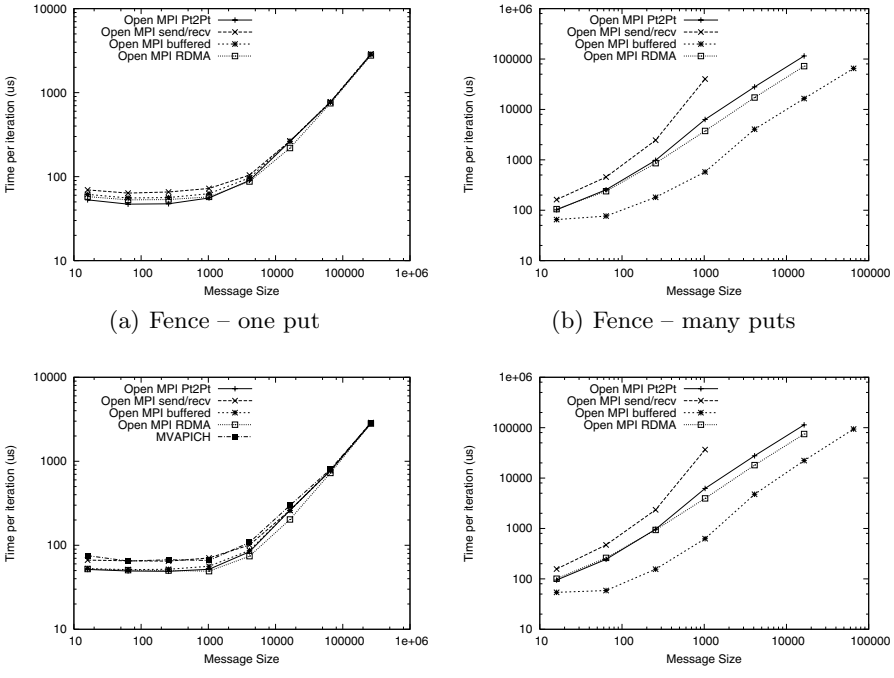


(a) Latency (b) Bandwidth

Fig. 4. Latency and Bandwidth of MPI_GET calls between two peers using generalized active synchronization

Again, the *send/recv* protocol suffers compared to the other protocols, due to the extra copy overhead compared to *rdma*, the extra transport headers compared to both *rdma* and *buffered*, and the delay in starting data transfer until the end of the synchronization phase. For large messages, where all protocols are utilizing RMA operations, realized bandwidth is similar for all implementations.

The latency and bandwidth of MPI_GET are shown in Fig. 4. The *rdma* protocol has lower latency than the send/receive based protocols, as the target process does not have to process requests at the MPI layer. The present *buffered* protocol does not coalesce reply messages from the target to the origin, so there is little advantage to using the *buffered* protocol over the *send/recv* protocol. For the majority of the bandwidth curve, all implementations other than the *rdma* protocol provide the same bandwidth. The *rdma* protocol clearly suffers from a performance issue that the MVAPICH2 implementation does not. For short messages, we believe the performance lag is due to receiving the data directly into the user buffer, which requires registration cache look-ups, rather than copying through a



(a) Fence – one put

(b) Fence – many puts

(c) Generalized – one put

(d) Generalized – many puts

Fig. 5. Ghost cell iteration time at 32 nodes for varying buffer size, using fence or generalized active synchronization

pre-registered "bounce" buffer. The use of a bounce buffer for MPI_PUT but not MPI_GET is an artifact of the BTL interface, which we are currently addressing.

Ghost Cell Updates. Fig. 5 shows the cost of performing an iteration of a ghost cell update sequence. The tests were run across 32 nodes, one process per node. For both fence and generalized active synchronization, the ghost cell update with large buffers shows relative performance similar to the put latency shown previously. This is not unexpected, as the benchmarks are similar with the exception that the ghost cell updates benchmark sends to a small number of peers rather than to just one peer. Fence results are not shown for MVAPICH2 because the tests ran significantly slower than expected and we suspect that the result is a side effect of the testing environment.

When multiple puts are intiated to each peer, the benchmark results show the disadvantage of the *send/recv* and *rdma* protocol compared to the *buffered* protocol. The number of messages injected into the MPI layer grows as the message buffer grows. With larger buffer sizes, the cost of creating requests, buffers, and the poor message injection rates of InfiniBand becomes a limiting factor. When using InfiniBand, the *buffered* protocol is able to reduce the number of messages injected into the network by over two orders of magnitude.

6 Summary

As we have shown, there are a number of implementation options for the MPI one-sided interface. While the general consensus in the MPI community has been to exploit the RMA interface provided by modern high performance networks, our results appear to indicate that such a decision is not necessarily clear-cut. The message coalescing opportunities available when using send/receive semantics provides much higher realized network bandwidth than when using RMA. The completion semantics imposed by a portable RMA abstraction also requires ordering that can cause higher latencies for RMA operations than for send/receive semantics.

Using RMA operations has one significant advantage over send/receive – the target side of the operation does not need to be involved in the message transfer, so the theoretical availability of computation/communication overlap is improved. In our tests, we were unable to see this in practice, likely due less to any shortcomings of RMA and more due to the two-sided nature of the MPI-2 one-sided interface. Further, we expected the computation/communication overlap advantage to become less significant as Open MPI develops a stronger progress thread model, allowing message unpacking as messages arrive, regardless of when the application enters the MPI library.

References

1. Geist, A., Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Saphir, W., Skjellum, T., Snir, M.: MPI-2: Extending the Message-Passing Interface. In: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124, pp. 128–135. Springer, Heidelberg (1996)
2. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI: The Complete Reference. the MPI-2 Extensions, vol. 2. MIT Press, Cambridge (1998)
3. Message Passing Interface Forum: MPI: A Message Passing Interface. In: Proc. of Supercomputing '93, IEEE Computer Society Press, pp. 878–883 (November 1993)
4. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The Complete Reference. MIT Press, Cambridge (1996)
5. Bonachea, D., Duell, J.: Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. Int. J. High Performance Computing and Networking 1(1/2/3), 91–99 (2004)
6. Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In: Proceedings of Supercomputing Symposium, pp. 379–386 (1994)
7. Booth, S., Mourao, F.E.: Single Sided Implementations for SUN MPI. In: Supercomputing (2000)
8. Trff, J.L., Ritzdorf, H., Hempel, R.: The implementation of mpi-2 one-sided communication for the nec sx-5. In: Supercomputing 2000, IEEE/ACM (2000)
9. Worringen, J., Gäer, A., Reker, F.: Exploiting transparent remote memory access for non-contiguous and one-sided-communication. In: Proceedings of ACM/IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002), Workshop for Communication Architecture in Clusters (CAC 02), Fort Lauderdale, USA (April 2002)

10. Argonne National Lab.: MPICH2, http://www-unix.mcs.anl.gov/mpi/mpich2/
11. Thakur, R., Gropp, W., Toonen, B.: Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication. Int. J. High Perform. Comput. Appl. 19(2), 119–128 (2005)
12. Huang, W., Santhanaraman, G., Jin, H.W., Gao, Q., Panda, D.K.: Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand. In: Int'l Sympsoium on Cluster Computing and the Grid (CCGrid), Singapore (May 2006)
13. Jiang, W., Liu, J., Jin, H.W., Panda, D.K., Buntinas, D., Thakur, R., Gropp, W.: Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (September 2004)
14. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104 (September 2004)
15. Shipman, G.M., Woodall, T.S., Graham, R.L., Maccabe, A.B., Bridges, P.G.: InfiniBand Scalability in Open MPI. In: IEEE International Parallel And Distributed Processing Symposium (to appear, 2006)
16. Woodall, T., et al.: Open MPI's TEG point-to-point communications methodology: Comparison to existing implementations. In: Proceedings, 11th European PVM/MPI Users' Group Meeting (2004)
17. Myricom, Inc.: Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet (2006)
18. Network-Based Computing Laboratory, Ohio State University: Ohio State Benchmark Suite, http://mvapich.cse.ohio-state.edu/benchmarks/

MPI-2 One-Sided Usage and Implementation for Read Modify Write Operations: A Case Study with HPCC*

Gopalakrishnan Santhanaraman, Sundeep Narravula, Amith. R. Mamidala, and Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University
Columbus, Ohio 43210
{santhana,narravul,mamidala,panda}@cse.ohio-state.edu

Abstract. MPI-2's One-sided communication interface is being explored in scientific applications. One of the important operations in a one sided model is *read-modify-write*. MPI-2 semantics provide MPI_Put, MPI_Get and MPI_Accumulate operations which can be used to implement *read-modify-write* functionality. The different strategies yield varying performance benefits depending on the underlying one-sided implementation. We use HPCC Random Access benchmark which primarily uses *read-modify-write* operations as a case study for evaluating the different implementation strategies in this paper. Currently this benchmark is implemented based on MPI two-sided semantics. In this work we design and evaluate MPI-2 versions of the HPCC Random Access benchmark using one-sided operations. To improve the performance, we explore two different optimizations: (i) software based aggregation and (ii) hardware-based atomic operations. We evaluate our different approaches on an InfiniBand cluster. The software based aggregation outperforms the basic one sided scheme without aggregation by a factor of 4.38. The hardware based scheme shows an improvement by a factor of 2.62 as compared to the basic one sided scheme.

1 Introduction

In the last decade MPI (Message Passing interface) [14] has evolved as the *de facto* parallel programming model in high performance computing scenarios. The MPI-2 standard provides a one-sided communication interface which is starting to be explored in scientific applications. One of the important operations in a one sided model is *read-modify-write*. Applications like [12] which is based on MPI-2 one-sided, predominantly use this operation. MPI-2 semantics provide MPI_Put, MPI_Get and MPI_Accumulate operations that can be used to implement the *read-modify-write* operations.

HPCC Benchmark suite is a set of tests that examine the performance of HPC architectures that stress different aspects of HPC systems involving memory and network

* This research is supported in part by Department of Energy's grant #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation's grants #CNS-0403342 and #CCF-0702675; grants from Intel, Sun Microsystems, Cisco Systems, and Linux Networx; Equipment donations from Intel, Mellanox, AMD, Apple, IBM, Microway, PathScale, Silverstorm and Sun Microsystems.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 251–259, 2007.

in addition to computation [3]. HPCC Random Access benchmark is one of the benchmarks in this suite which measures the rate of random updates to remote memory locations. Currently this benchmark is implemented based on MPI two-sided semantics. In this work we design different MPI-2 versions of the Random Access benchmark using the MPI-2 one-sided alternatives. We use the one-sided versions of the Random Access benchmark as a case study for studying different implementations of the *read-modify-write* operations and provide optimizations to improve the performance. In this work we use two different techniques: (i) software-based aggregation and (ii) hardware-based atomic operations provided by InfiniBand to improve the performance. We evaluate and analyze the benefits of using software aggregation using datatypes with one-sided operations as well as the hardware based direct accumulate approach.

The rest of the paper is organized as follows. In Section 2, we provide an overview of InfiniBand, MVAPICH2 and HPCC Random Access benchmark. In Section 3, we describe different strategies for implementing one-sided versions of the HPCC benchmark. In Section 4, we discuss the two optimization techniques that we propose. In Section 5, we show performance evaluations of the various schemes. We present some discussion in Section 6. In section 7, we present related work. Conclusions and future work are presented in Section 8.

2 Background

In this section, we provide a brief background on InfiniBand, MVAPICH2 and the HPCC Random Access benchmark.

InfiniBand: The InfiniBand Architecture (IBA) [7] is an industry standard. It defines a switched network fabric for interconnecting processing nodes and I/O nodes. IBA supports both channel semantics and memory semantics. In channel semantics, send/receive operations are used for communication. In memory semantics, InfiniBand provides Remote Direct Memory Access (RDMA) operations, including RDMA write and RDMA read. InfiniBand also provides RDMA atomic operations namely fetch_and_add and compare_and_swap. The network interface card on the remote node guarantees the atomicity of these operations. The operations are performed on 64 bit values. Leveraging this atomic fetch_and_add mechanism is one of the focus of this paper.

MVAPICH2: MVAPICH2 is a popular implementation of MPI-2 over InfiniBand[1]. The implementation is based on MPICH2. MPICH2[2] supports MPI-1 as well as MPI-2 extensions including one-sided communication. In MVAPICH2 the one sided implementation of MPI_Put and MPI_Get uses the InfiniBand RDMA_Write and RDMA_Read services to provide high performance and scalability to the applications. The MPI_Accumulate is currently two sided based in the sense that the remote node is involved in performing the accumulate operation on that node. In this work we also provide a prototype of truly one-sided implementation of the MPI_Accumulate operation using the InfiniBand atomic fetch_and_add operation.

HPCC: The HPC Challenge (HPCC) benchmark suite has been funded by the DARPA High Productivity Computing Systems (HPCS) program to help define the performance boundaries of future Petascale computing systems [5]. HPCC is a suite of tests that

examine the performance of high-end architectures using kernels with memory access patterns more challenging than those of the High Performance LINPACK (HPL) benchmark used in the Top500 list. The Random Access benchmark measures the rate of integer updates to random memory locations (GUPs). It uses xor operation to perform the updates on the remote node. It allows optimization in terms of aggregating up to 1024 updates to improve the performance.

3 One Sided HPCC Benchmark: Design Alternatives

In this section we describe different approaches taken to implement the one-sided version of the HPCC Random Access benchmark. As described earlier, random access benchmark measures the GUPs rating. An update is a read modify write operation on a table of 64-bit words. An address is generated, the value at that address read from memory, modified by an xor operation with a literal value and that new value is written back to memory. Currently the MPI version of the benchmark is a two sided version.

3.1 Design Issues

In this section we first describe the semantics and mechanisms offered by MPI-2 for designing one-sided applications. In a one-sided model, the sender can access the remote address space directly without an explicit receive posted by the remote node. The memory area on the target process that can be accessed by the origin process is called a Window. In this model we have the communication operations MPI_Put, MPI_Get and MPI_Accumulate and the synchronization calls to make sure that the issued one sided operations are complete. There are two types of synchronization: a) active in which the remote node is involved and b) passive in which the remote node is not involved in the synchronization. The active synchronization calls are collective on the entire group in case of MPI_Fence or a smaller group in case of Start_Complete and Post_Wait model. This could lead to some limitations when the number of synchronizations needed per process are different for different nodes. In passive synchronization the origin process issues MPI_Lock and MPI_Unlock call to indicate the beginning and end of the access epoch. Next we describe our approach taken in designing the one-sided versions of the HPCC Random Access benchmark. We map the table memory to the Window so that the one-sided versions can read and write directly to this memory.

3.2 HPCC Get-Modify-Put (HPCC_GMP)

In the first approach we call MPI_Get to get the data, perform the modification, then use MPI_Put to put the updated data to the remote location. As compared to the two sided versions there are no receive calls made on the remote node. Also the active synchronization model cannot be used since we cannot match the number of synchronization calls across all nodes. This is because the number of remote updates as well as the location of the remote updates for each node can vary randomly. Hence we use passive synchronization MPI_Lock and MPI_Unlock calls in this scheme. Further we need one set of Lock and Unlock calls to fetch the data, perform the modification, then another

set of Lock and Unlock operations to put the data. The reason for this is the flexibility of MPI-2 semantics which allows MPI_Get to fetch the data in Unlock. Also the MPI_Get and MPI_Put can be reordered within an access epoch. We describe this approach in Fig. 1a and will henceforth refer to it as *HPCC_GMP*. This approach leads to a lot of network operations resulting in lower performance. Further the possibility of incorrect updates increases. This is due to the coherency issues that might arise because of parallel updates occurring simultaneously.

3.3 HPCC Accumulate (HPCC_ACC)

Our next approach uses the MPI_Accumulate operation provided by MPI-2. MPI-2 semantics provide MPI_Accumulate which are basically atomic reductions. This non collective one-sided operation combines communication and computation in a single interface. It allows the programmer to update atomically remote locations by combining the content of the local buffer with the remote memory buffer. This implementation calls MPI_Accumulate between MPI_Lock and MPI_Unlock synchronization calls. Using this approach shown in Fig. 1b, we do not have the issue of incorrect updates. Also as compared to our *HPCC_GMP*, the number of network operations is significantly reduced. Another approach is to use Accumulate with Active synchronization model using Win_Fence. This could be done by calling Win_Fence at the very beginning, performing all the updates using MPI_Accumulate and then call one Win_Fence at the very end. All the processes need to call two Win_Fence calls, one at the beginning and one at the end. However since MPI-2 semantics allows the actual data transfer to occur inside the synchronization call that closes the exposure epoch, all the accumulates could happen during the second Win_Fence call. Many MPI implementations actually make use of this flexibility. This violates the random benchmark rule that you could store only 1024 updates at the maximum. Hence we did not consider this approach.

4 Optimizations

In this section we describe two optimizations we propose in this paper to improve the performance of the one-sided version of HPCC Random Access benchmark.

4.1 HPCC Accumulate with Software Aggregation (HPCC_ACC_AGG)

In this technique we want to aggregate or pack a number of update operations together so that the overhead of sending as well as synchronization operations can be reduced. Using this approach, we aggregate a bunch of update operations before sending them as a single communication operation. The HPCC random access benchmark allows each processor to store up to 1024 updates before sending them out. The MPI-2 semantics provides *datatypes* feature that can be leveraged to achieve aggregation. For one-sided operations both the sender and destination datatypes need to be created. We create MPI_Type_struct sender and receiver datatypes to represent a bunch of updates. We then use the created datatypes to issue a single communication call as shown in Fig. 1c. Using this approach we expect to improve the performance since the number of network operations are minimized.

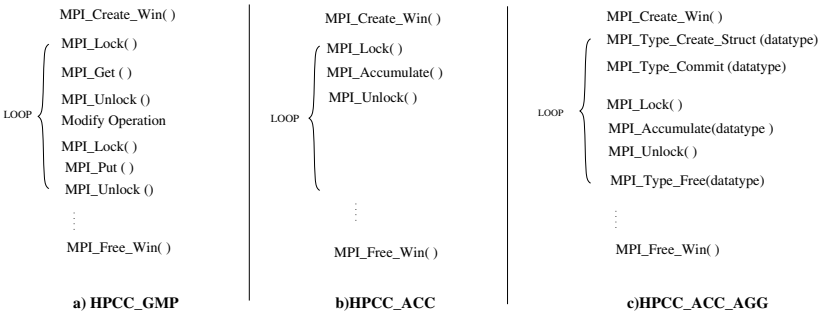


Fig. 1. Code snippets of one-sided versions of HPCC Random Access benchmark

4.2 Hardware Based Direct Accumulate (HPCC_DIRECT_ACC)

InfiniBand provides hardware atomic fetch and add operation that can be leveraged to optimize MPI_Accumulate operation for MPI_SUM. The Accumulate operations use the hardware fetch and add operation that can provide good latency and scalability. One limitation of this approach is that we can only do single 64 bit accumulates with each fetch and add operation, i.e. aggregation is not possible. A benefit of using this approach is that since it is truly one-sided in nature, it provides more scope for overlap that can lead to improved performance. It is to be noted that this optimization is implemented in the underlying MVAPICH2 MPI library as a prototype and is transparent to the application writer.

5 Performance Evaluation

In this section, we evaluate the performance of one-sided version of the HPCC benchmark for the different schemes. We present some micro-benchmark results to give the basic performance of different one-sided operations and show the potential of our proposed optimizations. The experimental testbed is x86 64 node cluster with 32 Opteron nodes and 32 Intel nodes. Each node has 4GB memory and equipped with PCI-Express interface and InfiniBand DDR network adapters (Mellanox InfiniHost III Ex HCA).

5.1 Basic Performance of One-Sided Operations

In this section we show the performance of the basic one-sided operations MPI_Put, MPI_Get and MPI_Accumulate. Fig. 2a shows the small message latency for these operations. The latency for 8bytes for put and get are 5.68 and 11.03 usecs, respectively, whereas the accumulate latency is 7.06 usecs. Since get_modify_put implementation needs both get and put in addition to modify and synchronization operation, we expect this performance to be lower compared to the accumulate based approach.

5.2 HPCC One-Sided Benchmark Performance with Different Schemes

In this section we evaluate the performance of the two different versions of the benchmark *HPCC_GMP* and *HPCC_ACC*. The results are shown in Fig. 2b. As expected

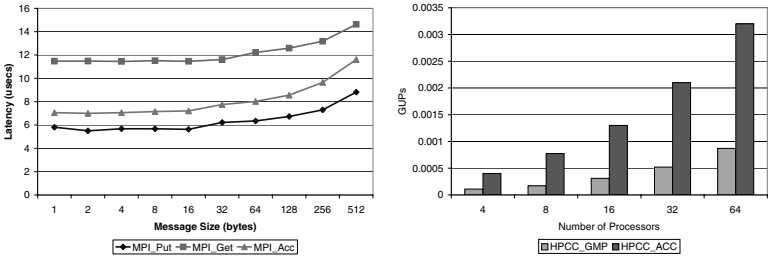


Fig. 2. Basic Performance (a) Micro-benchmarks and (b) Basic HPCC GUPs

the *HPCC_ACC* performs better than the *HPCC_GMP* because of the number of synchronization and communication operations in *HPCC_GMP*. The overhead of these additional network operations leads to lower performance of *HPCC_GMP*. This performance gap increases with increasing number of processors since the synchronization cost increases further for larger number of nodes. Hence we choose *HPCC_ACC* as our base case for further optimizations and evaluations.

5.3 Aggregation Benefits

To improve the performance of Accumulate operation, we proposed aggregation using Accumulate with datatype. We evaluate the performance benefits of using datatype at micro-benchmark level. In basic version we do multiple accumulates corresponding to the number of updates. In aggregated version we create a datatype corresponding to the number of updates and perform a single accumulate operation with that datatype. Fig. 3a shows the results of our study. With increasing amounts of aggregation, the Accumulate with datatype outperforms the multiple accumulate schemes. With aggregation the cost of sending overhead and the synchronization overheads are limited to the number of aggregated operations. Next we compare the performance of *HPCC_ACC_AGG* with *HPCC_ACC* for 512 and 1024 aggregations. The results are shown in Fig. 3b. We observe a similar trend with the optimized *HPCC_ACC_AGG* performing better than the *HPCC_ACC* scheme. This result demonstrates the benefits of aggregation.

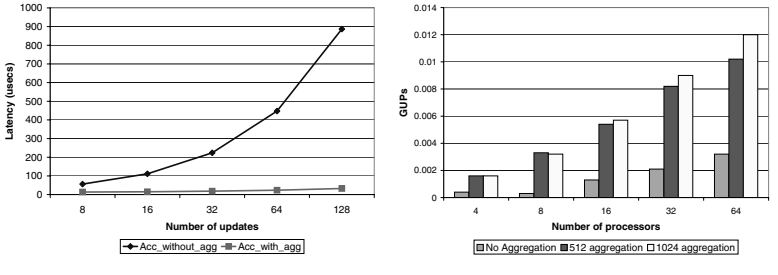


Fig. 3. Aggregation Performance Benefits (a) Basic Aggregation Micro-benchmarks and (b) HPCC with Aggregation

5.4 Hardware Based Direct Accumulate

In this section we first study the benefits that could be achieved using the hardware based fetch and add operation to implement a read modify write operation at microbenchmark level (*DIRECT_ACC*). We compare its performance with the the schemes that uses Get Modify Put (*GMP*) approach and MPI_Accumulate (*ACC*) approach. The MPI implementation allows optimizations that delays the actual lock and data transfer operation to happen during unlock. In this case measuring just the lock and unlock cost does not provide any additional insight. Hence we measure the latency that includes both data transfer and lock/unlock synchronization operation. Fig. 4a compares the basic performance of *GMP*, *ACC* and *DIRECT_ACC*. We note that for single updates of 64bit integer, the (*DIRECT_ACC*) scheme provides the lowest latency. This is because the existing MPI_Accumulate implementation is inherently two sided whereas the Direct Accumulate implementation makes use of the truly one-sided hardware feature.

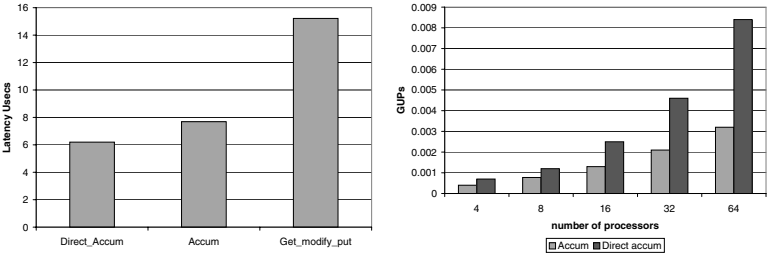


Fig. 4. Direct Accumulate Performance Benefits (a) Micro-benchmarks and (b) HPCC with Direct Accumulate

Next we try to understand the benefits that a hardware based Accumulate operation can provide to an application. To evaluate this we modify the *HPCC_ACC* benchmark to use the MPI_SUM operation instead of the MPI_BXOR operation and call this as *HPCC_ACC_MOD*. The verification phase is correspondingly modified. We then compare the *HPCC_ACC* which uses the existing MPI_Accumulate implementation in the MVAPICH2 library with the modified *HPCC_ACC_MOD* which uses our Direct Accumulate prototype implementation. The results are shown in Fig. 4b. We observe that the Direct accumulate performs significantly better than the basic accumulate. Also the Direct Accumulate seems to scale very well with increasing number of processors. The reason for this is two-fold: 1) low software overhead and 2) true one-sided nature of the hardware based Direct Accumulate.

Finally we compare our two proposed techniques Direct Accumulate and software aggregation (Accumulate with datatype). The results are shown in Fig. 5. The software aggregation scheme beats the hardware based direct accumulate approach since currently the hardware fetch and add operation does not support aggregation. Also the gap between the two schemes seem to be narrowing with increasing nodes. This demonstrates the scalability of the hardware based operations and suggests the benefits of having aggregation in hardware as well.

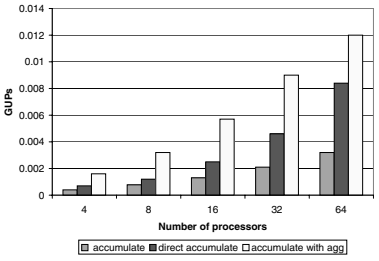


Fig. 5. Software Aggregation vs Hardware Direct Accumulate benefits

6 Discussion

Current implementations for HPCC Random Access benchmark are based on two-sided communication primitives. While the main objective of this paper is not to compare the designs based on one-sided and two-sided semantics, it is also important in this context to note that the current one-sided implementations are largely based on two-sided primitives in the MPI libraries and hence, such an evaluation is not as informative. InfiniBand's hardware fetch and add operation provides a design opportunity for a Direct Accumulate for MPI_Sum operation for a single 64 bit field. While we have demonstrated that both aggregation and direct hardware based accumulation has benefits, an aggregated direct accumulate is likely to yield much higher performance benefit. However it is clearly not possible to implement such a design with current hardware.

7 Related Work

There are several studies regarding implementing one sided communication in MPI-2. Some of the MPI-2 implementations which implement one sided communication are MPICH2 [2], WMPI [**?**], SUN-MPI [3],In [10,8], the authors have used InfiniBand hardware features to optimize the performance of MPI-2 one sided operations. Other researchers [9] study the different approaches for implementing the one sided atomic reduction. The authors in [**?**] have looked at utilizing the hardware atomic operations in Myrinet/GM to implement efficient synchronization operations. Recently several researchers have been looking at providing optimizations to the HPCC benchmark. In [6] the authors have suggested techniques for optimizing the Random access benchmark for Blue Gene clusters.

8 Conclusions and Future Work

In this paper we designed MPI-2 one sided versions of HPCC random access benchmark using get_modify_put and MPI_Accumulate operations . We evaluated these two different approaches on a 64 node cluster. To improve the performance we explored two techniques: a) software based aggregation and b) utilizing hardware atomic operations.

We analyzed the benefits and trade-offs of these two approaches. Our studies show that the software based aggregation performs the best. We also demonstrate the potential and scalability of the hardware based approach. As part of future work we would also like to evaluate the potential benefits with one sided applications which use these operations.We also plan to contribute our one-sided versions of the benchmark to the HPCC benchmarking group.

References

1. Network Based Computing Laboratory, MVAPICH2 http://mvapich.cse.ohio-state.edu/
2. Argonne National Laboratory: MPICH2 http://www-unix.mcs.anl.gov/mpi/mpich2/
3. Booth, S., Mourao, F.E.: Single Sided MPI Implementations for SUN MPI. In: Supercomputing (2000)
4. Buntinas, D., Panda, D.K., Gropp, W.: NIC-Based Atomic Remote Memory Operations in Myrinet/GM. In: Workshop on Novel Uses of System Area Networks (SAN-1) (February 2002)
5. Dongarra, J., Luszczek, P.: overview of the hpc challenge benchmark suite. In: SPEC Benchmark Workshop (2006)
6. Garg, R., Sabharwal, Y.: Optimizing the HPCC randomaccess benchmark on blue Gene/L Supercomputer. ACM SIGMETRICS Performance Evaluation Review (2006)
7. InfiniBand Trade Association: InfiniBand Architecture Specification, Release 1.0 (October 24, 2000)
8. Jiang, W., J.Liu, Jin, H.W., Panda, D.K., Buntinas, D., R.Thakur, W.Gropp: Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. EuroPVM/MPI (September 2004)
9. Nieplocha, J., Tipparaju, V., Apra, E.: An evaluation of two implementation strategies for optimizing one-sided atomic reduction. In: IPDPS (2005)
10. Liu, J., Jiang, W., Jin, H.W., Panda, D.K., Gropp, W., Thakur, R.: High Performance MPI-2 One-Sided Communication over InfiniBand. In: CCGrid 04 (April 2004)
11. Mourao, F.E., Silva, J.G.: Implementing MPI's One-Sided Communications for WMPI. newblock In: EuroPVM/MPI (September 1999)
12. Thacker, R.J., Pringle, G., Couchman, H.M.P., Booth, S.: Hydra-mpi: An adaptive particle-particle, particle-mesh code for conducting cosmological simulations on mpp architectures. High Performance Computing Systems and Applications (2003)
13. HPCC Benchmark Suite. http://icl.cs.utk.edu/hpcc
14. Gropp, W., Lusk, E., Skjellum, A.: Using MPI:Portable Parallel Programming with the Message Passing Interface, 2nd edn. MIT Press, Cambridge, MA (1999)

RDMA in the SiCortex Cluster Systems

Lawrence C. Stewart, David Gingold, Jud Leonard,
and Peter Watkins

SiCortex, Inc.
larry.stewart,david.gingold,jud.leonard,
peter.watkins@sicortex.com

Abstract. The SiCortex cluster systems implement a high-bandwidth, low-latency interconnect. We describe how the SiCortex systems implement RDMA, including zero-copy data transfers and user-level networking. The system uses optimistic virtual memory registration without page locking. Finally, we provide preliminary performance results.

1 Introduction

In a compute node of a typical cluster system, the network interface is a separate device on the I/O bus. In the SiCortex system, the NI is integrated on-chip and is coherent with the multicore CPU caches. This high level of integration permits increased packaging density, reduced power consumption, greater reliability, and improved interconnect performance.

We took the opportunity to rethink the NI's design at all levels, considering together the hardware, operating system, and communication software. We came to the conclusions that letting the application talk directly to the NI—so-called *user-level networking*, or *OS-Bypass*—and supporting DMA access to virtually addressed application buffers, so-called *zero-copy*, were essential. These techniques are not new, but we have been more aggressive than previous efforts in our use of optimistic methods.

2 Related Work

Many groups have studied interconnect APIs for high-performance networking, including [1,2,3,4,5,6,7,8]. Generally, workers in the field have come to the conclusion that permitting user-level access to the network hardware reduces latency for short messages, and that eliminating memory-to-memory copies—so-called zero-copy or minimal-copy [9]—can reduce CPU overheads and deliver greater bandwidth for large messages. A number of systems have been built on these principles [3,10,11,12].

2.1 User-Level Networking

In user-level networking, application code running in user mode has direct access to the network hardware, so that it can initiate operations without incurring system call overheads.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 260–271, 2007.

In user-level networking designs, the application libraries directly command I/O operations. While this avoids system calls, it can also introduce security problems and requires a means of sharing the device hardware among applications and that the NI be able to route incoming traffic to the correct application.

The implementation challenges include providing virtual-to-physical address translation for the hardware, and providing user-mode access to hardware in a way that does not compromise the integrity of the kernel or other processes.

A good introduction is [4], which introduces Application Device Channels, as protected communication paths from user code to hardware. Our design is similar; we use the VM system to map sets of device registers. But instead of mapping device memory, we share data structures in main memory.

2.2 Zero-Copy

In zero-copy I/O, an application buffer is handed directly to the device [13]. At the receiver, data is written by the hardware directly into application buffers.

Zero-copy is primarily a tool for improving bandwidth and reducing CPU overhead for large messages. The QLogic InfiniPath [14] achieves good results by copying all data but most devices use DMA. (For a contrarian view, see [15].)

The previous work closest to ours is [5] and [9]. These efforts use device address translation tables managed by user code, but memory is pinned and the wire messages carry virtual addresses [9] or are inspected by receive-side code [5]. In contrast, we don't pin memory, and wire messages carry translation table indices.

Most previous work focused on locking memory during DMA operations so that the OS would not unmap or reallocate memory in use by the hardware [16,17,18,12].

Previous designs also guarantee the success of DMA operations. Locally, this requires pinned memory and preloaded device translations or that translation misses be handled in real time. One-sided operations use advance window setup as in MPI-2, hardware translation [12], or RPC setup [16].

The Optimistic Direct Access File System (OADFS) [19,20,21] comes closest to our optimistic registration approach. In ODAFS, clients initiate RDMA commands using old cached addresses that may no longer be valid. If the server translation fails, a remote exception is raised, but a fault can occur only before the transfer starts. Once underway, a DMA operation is guaranteed to finish successfully.

3 The SiCortex Hardware

The SiCortex system [22] is a purpose-built Linux-based cluster computer, with up to 972 compute notes. Each node, shown in Figure 1, is a six-way symmetric multiprocessor (SMP) with coherent caches, interleaved memory interfaces, high speed I/O, and a programmable interface to the interconnect fabric. Each node is one system-on-a-chip plus external DRAM. Each node runs a single SMP Linux kernel.

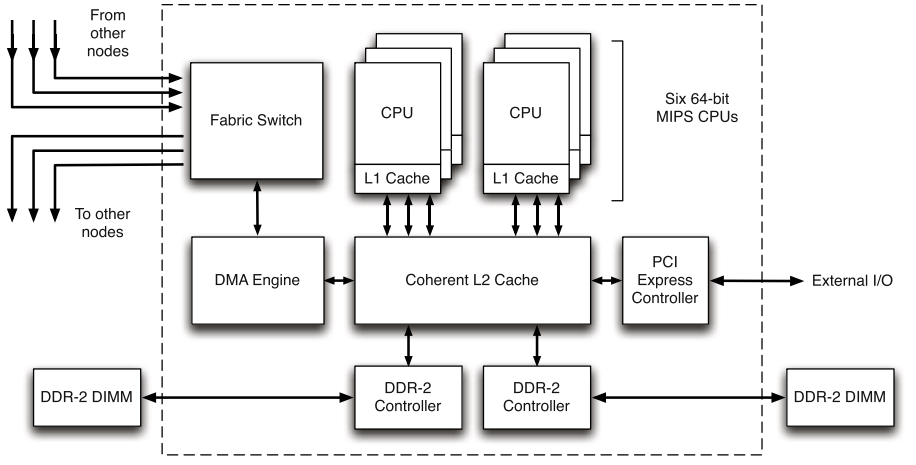


Fig. 1. Cluster Node, including DMA Engine, Switch, and Links

The interconnect fabric consists of three components (see Figure 1): the DMA Engine, the fabric switch, and the fabric links. The DMA Engine implements the software interface to the fabric. The switch forwards traffic between incoming and outgoing links, and to and from the DMA Engine. The links connect directly to other nodes in the system [23]. The links run at 2 GB/s. The switch is a virtual-channel cut-through buffered crossbar switch [24].

4 The Hardware-Software Interface

In an approach similar to [4], the network interface appears to user-mode code as a set of in-memory data structures shared with the DMA Engine and with a small set of memory-mapped registers. The bulk of an application's interaction with the DMA Engine happens through shared memory. The data structures include:

- a *command queue* (CQ), a circular memory buffer which a CPU writes, providing commands for the DMA Engine to execute;
- an *event queue* (EQ), a circular memory buffer into which the DMA Engine writes incoming short messages and events indicating DMA completion;
- a *heap*, where the CPU can place command chains, and to which the DMA can write additional messages;
- a *route descriptor table* (RDT), whose indices serve as handles specifying routes through the fabric to remote DMA contexts; and
- a *buffer descriptor table* (BDT), whose indices serve as handles specifying areas in the application's virtual memory.

Similar to [5] and [9], in order to enforce proper OS protection, only the kernel may write a process's BDT and RDT tables. Application software detects incoming

events by polling or interrupts. The command and event queues are implemented as ring buffers, with read and write pointers stored in device registers.

4.1 DMA Primitives

Software issues DMA commands by writing them to the command queue and then writing a DMA Engine register. The available commands include:

- *send-event*: Deliver data in a single packet to a remote EQ.
- *write-heap*: Write data in a single packet to a location in a remote heap.
- *send-command*: Transmit an embedded DMA command in a packet to be executed by the remote context's DMA Engine.
- *do-command*: Decrement a counter, and if the result is negative, execute a specified list of commands in the local heap.
- *put-buffer*: Transmit an entire memory segment to a remote context's memory. Upon completion, optionally generate a remote event or execute remote commands.

The ability to remotely execute commands provides much of the power of these primitives. Software implements RDMA GET operations, for example, by issuing one or more send-command commands, each with an embedded put-buffer command.

We targeted this design for MPI but we believe it is also well suited to other networking systems, such as TCP/IP, SHMEM, and GASnet [25,26,16].

4.2 The Put-Buffer Command

The put-buffer command is the mechanism that applications use to achieve zero-copy RDMA.

The put-buffer command specifies memory locations as a *buffer descriptor* (BD) index and offset pair. The BD index represents an entry in a context's BD table; user-level software has associated that entry with a region of its virtual memory (as described in Section 5). The offset is a byte offset within this memory region. A *route handle* field in the command identifies the operation's destination node and context. The destination BD index references an entry in the BD table of the destination context.

The DMA Engine signals the normal completion of the command by posting a notifier event to the EQ of the destination context. The command can also specify commands to be executed by the DMA Engine at the destination.

5 DMA Registration

Virtual memory presents a challenge for RDMA implementations, both because of the need to translate from virtual to physical addresses and because a valid virtual page at a given moment may not be mapped to physical memory at all. RDMA designs often solve this by requiring that virtual pages be pinned

to physical pages during the lifetime of RDMA operations. But this solution is awkward. In MPI, for example, pages that are pinned when the application initiates a send operation might need to remain so for an arbitrarily long time before the application posts a matching receive.

Our implementation requires that virtual pages used for RDMA be registered, but not pinned. Registration creates an association between an application's virtual page and the corresponding physical page. The OS invalidates this association when it unmaps the virtual page. An RDMA operation which references an invalidated association results in a fault, requiring that software re-start the operation.

We call this scheme *optimistic registration* because, in practice, the operating system should rarely unmap the virtual pages of a running application that fits in memory. Optimistic registration is efficient in this common case, simple to implement, and allows the OS the flexibility to unmap pages when it needs to do so.

5.1 Implementing Registration

Our DMA Engine uses entries in a Buffer Descriptor Table (BDT) to associate pages of virtual memory with physical memory. The BDT is an in-memory data structure, and there is one BDT per DMA context. A BDT entry stores the physical address associated with a single page of virtual memory. User-level code directs the kernel to map virtual addresses to specified BDT entries. The BDT is managed by user-level code, but written only by the kernel. When the kernel unmaps a virtual address, it invalidates corresponding BDT entries.

When user-level code issues an RDMA operation, in the form of a put-buffer command, to the DMA Engine, it specifies a memory location as an index in the BDT and a byte offset within that page. The user-level code can pass its BDT indices to other nodes.

A single BDT entry is physically a 64-bit word that stores a physical memory address, a size, a valid bit, and a read-only bit. In our implementation, a user-level BDT entry maps a single 64KB virtual page in its entirety.

5.2 User-Level BDT Management

The BDT performs a function similar to that of a traditional virtual memory page table, but it differs in that user-level code directs exactly how the BDT entries are used. Different applications can choose different management schemes.

When an MPI call needs to initiate an RDMA operation (either at the sending or receiving end), the library uses a red-black tree to determine whether the associated virtual addresses are already mapped to BDT entries. If the addresses are not mapped, the software allocates BDT entries, invokes a system call to map them, and inserts the information into the red-black tree.

5.3 Kernel-Level BDT Management

The kernel provides system call interfaces to allow user-level code to map BDT entries. The registration kernel call is lightweight. It maps the virtual address

to a physical address by traversing the process page tables, with access and protection checks, and then marks the physical page as being used by DMA. If the access permits writes, the page is proactively marked dirty[1]. The page is not pinned, and no data structures are allocated.

When the kernel invalidates a virtual mapping, it must invalidate associated BDT entries, just as it invalidates its page table entries. If there is a valid BDT mapping, the kernel must clear the BDT entry, invalidate associated cached entries within the DMA Engine, and ensure that any pending memory operations complete before reclaiming the physical page.

5.4 BDT Faults

RDMA operations complete successfully if the BDT entries they reference remain valid until the DMA transfer finishes. Upon completion, the DMA Engine writes a notifier event to the receiving context. There is no need to unregister BDT entries.

An RDMA operation that references an invalidated BDT entry results in a *buffer descriptor fault*. The fault might happen at either the sending or receiving end. All faults generate a notifier event at the receiving end.

If a buffer descriptor fault is detected at the transmit node, the transmit DMA Engine abandons the rest of the transfer, and sends a LASTDMA packet with a fault indication.

If a buffer descriptor fault is detected at the receive node, the receive DMA Engine sets a bit in a per context bitmap indexed by the *notifier* transfer ID and discards the packet. There is no way for the receive end to cause the transmit end to abort the transfer, so the receive DMA Engine continues to process (and discard) packets.

When the LASTDMA packet arrives, the receive DMA Engine checks the error bitmap. If the LASTDMA packet does not report a transmit fault, and the local bitmap does not record a receive fault, then the DMA Engine generates a normal completion event. If a fault was reported, the receive node DMA Engine generates a local fault event.

User-level software, not the kernel, recovers from a buffer descriptor fault. The software does this by remapping the BDT entries referenced by the operation and then restarting the RDMA.

6 MPI

Our MPI implementation is derived from the MPICH2 software from Argonne National Laboratory. At present, our implementation supports all MPI-1 and selected MPI-2 features.

6.1 MPI Internals

The MPI software sends small messages using an eager protocol that copies the data at both ends, and larger messages using a rendezvous protocol that uses

[1] Pages marked copy-on-write must be pre-faulted.

RDMA for zero-copy transfers. The receivers perform MPI matching in software, using optimized code to traverse posted receive and early send queues.

Longer messages are sent using a rendezvous protocol. The sending end transmits a rendezvous-request message, including the source application buffer's BDT indices. At the receiving end, the MPI library matches the request, and resolves the destination buffer's address to BDT entries. The receiver initiates the RDMA by issuing a send-command operation to the DMA Engine. The send-command encodes a put-buffer command, which is sent to the sender's DMA Engine.

The sending DMA Engine, having received the remote command, executes it by looking up the specified BDT entry's physical address, reading memory, and sending a series of DMA packets to the receiver. The receiving DMA Engine looks up the receive BDT entry's physical address and writes the packets' payloads to memory. (An invalid buffer descriptor at either end would be treated as described in Section 5.2.)

At the end of the transfer, the receiving DMA Engine posts a notifier event to the receiver's event queue to signal that the put-buffer is complete. The receiver's MPI library then sends an event to the sender, notifying it both that it is done with the sender's BDT entries and that the send operation is complete.

For messages larger than 64KB, the MPI library breaks the operation into multiple 64 KB *segments*. It schedules a few segments along each of the fabric's three independent paths available from source to destination, then schedules additional segments as early ones complete. This increases end-to-end bandwidth by load-sharing across available paths, and helps to avoid hot-spots. The technique also hides the DMA registration overhead in a way similar to the pipelined pinning approach described in [18].

7 Performance

We present latency and bandwidth results for MPI ping-pong with particular emphasis on long messages, which use the rendezvous protocol with RDMA. These results are preliminary; we expect improvements with future revisions of software, microcode, and hardware.

We have optimized the critical code path for short messages. With our approach, we have demonstrated a node-to-node short-message MPI latency (in a ping-pong test) of 1420 nanoseconds. Of this, the DMA Engine and fabric spend 950 nanoseconds performing the send-event operation. The remaining 470 nanoseconds—just 235 processor clock cycles at 500 MHz–are the software overhead of the MPI implementation.

Figure 2 shows overall one-way latency and bandwidth for various size MPI messages in a ping-pong test. We transition to RDMA above 1024 bytes. Off-node bandwidth is better than on-node because the on-node case loads the node's memory system with both reads and writes. Above 128 KB the software uses multiple interconnect paths.

Figure 3 shows latency for rendezvous mode transfers. For a 2K-byte transfer, the overall latency is 10.9 microseconds. Of this, 4.3 microseconds are in the

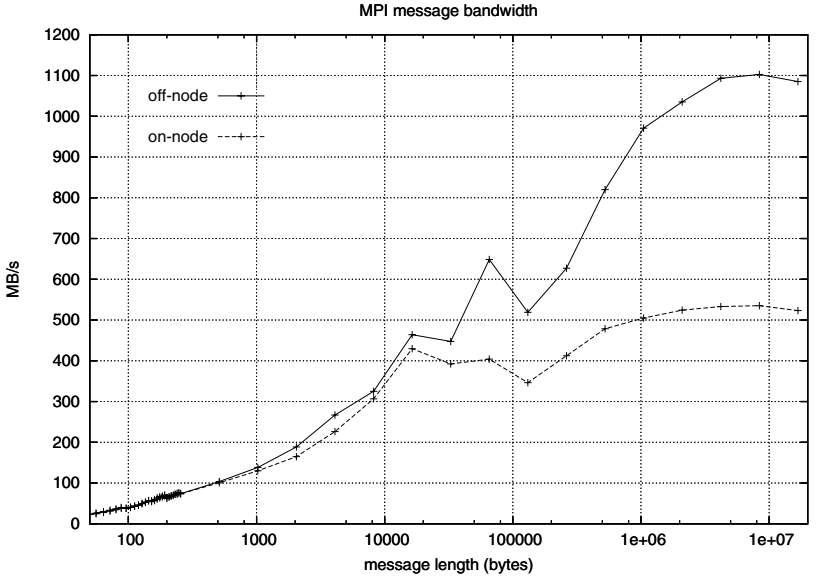


Fig. 2. MPI Ping-Pong Bandwidth

MPI software. The DMA Engine and fabric are responsible for the remaining 6.6 microseconds: 1.2 transmitting the rendezvous request from node to node with a send-event command, and 5.4 microseconds for the RDMA get operation, which includes 3.4 microseconds of fixed overhead.

In the tests that generated the above data, the cost of DMA registration is amortized over many iterations of send and receive operations. Measured deliberately, the cost of registering and de-registering a single 64K-byte page of memory is about 7 microseconds. By comparison, locking and unlocking a 64K-byte page of memory, via the `mlock()` and `munlock()` Linux system calls, also requires about 7 microseconds in this system.

Figure 4 shows results from the HPC Challenge Random Ring benchmark on an SC648 with 108 six-CPU nodes. The figure shows random ring latency and bandwidth results for all 108 nodes active, and with one to six CPUs per node active. HPCC Ping-Pong Latency is shown as well. The bandwidth results reflect an aggregate bandwidth limit around 1200 MB/sec shared by transmit and receive activities of all processors.

7.1 Effects of Registration

The benefits that we expect from optimistic registration are largely gains in flexibility rahter than gains in performance. Here we argue why we expect optimistic registration to perform at least as well as a page pinning approach, and explain where we see its benefits in flexibility.

RDMA implementations that require page pinning typically maintain a “pin cache” to limit the number of pages that they may have pinned at one time [17].

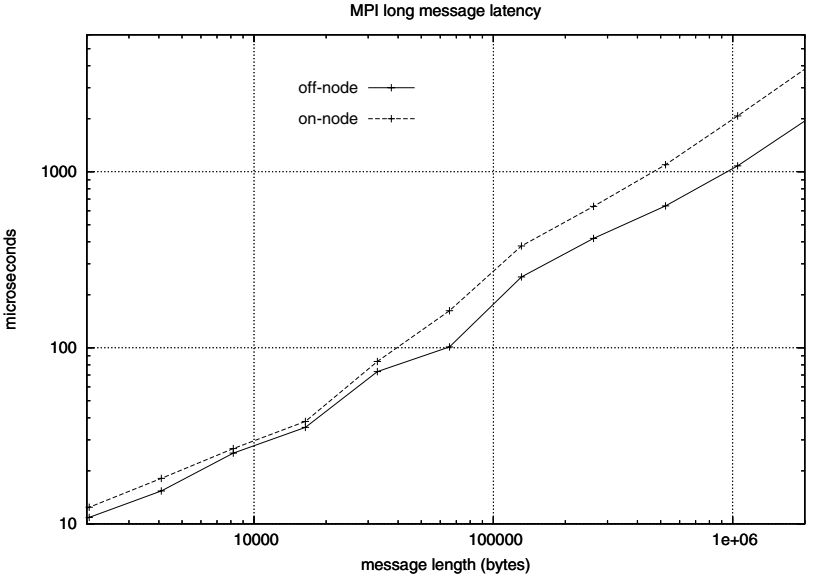


Fig. 3. MPI Ping-Pong Latency

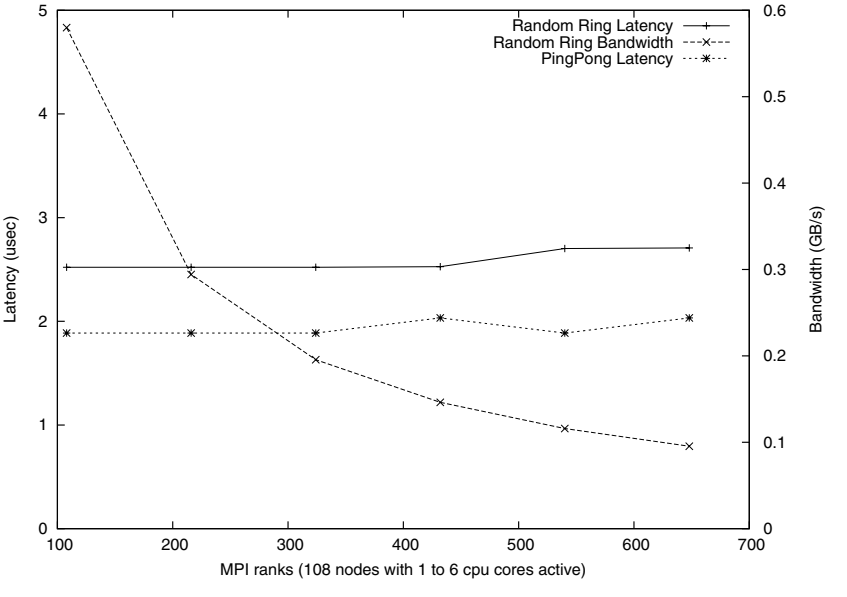


Fig. 4. HPCC RandomRing Results

When the application's communication working set is smaller than the pin cache, there is very little pinning and unpinning. When it exceeds the cache size, the cache will thrash. Maintaining a limited-size pin cache prevents applications from starving the system of usable physical memory.

In some respects, our optimistic registration scheme is similar to such a pin cache: the first-time communication cost of registering a page is roughly that of pinning a page, and our communication library avoids this subsequently by noticing that the memory has been registered previously. But in our system, the OS does not limit the number of pages an application may have registered. *It is as if we are operating with a pin cache that has an unlimited size.*

Applications that perform well in a pin cache system clearly should also perform well under optimistic registration. Applications that suffer because of an insufficiently large pin cache should do better under optimistic registration. But the real benefit is not the difference in usable memory size, but simply that the system can run without having the bulk of its memory pinned. The operating system can page out an idle application, and applications can take advantage of backing store, even for registered memory pages.

8 Conclusion

In the SiCortex system, applications manage the network interface's buffer descriptor table which associates virtual addresses with physical addresses. The kernel creates these associations as directed by user-level code, but can invalidate them as it needs. DMA transfers, even those that have already started, can receive translation faults at any time, which are reported to user mode software for recovery. This approach avoids pinning in the critical latency path, in trade for more expensive, but infrequently used, recovery paths in the event of faults.

The performance effect of this design is similar to the use of a cache of pinned memory, but with an unlimited cache size.

9 Future Work

The present paper is a preliminary report on certain aspects of communication hardware and software in the SiCortex system. The current software implementation is sufficient to run MPI-1 applications, but could be enhanced in many ways.

Collective operations are not yet optimized, although a number of DMA Engine features were designed for this purpose. We expect MPI-2 one sided operations to be straightforward, but they aren't done yet. Our implementation doesn't provide asynchronous progress, and so reduces the available overlap of communication and computation [27]. We have not as yet implemented non-MPI communications APIs such as SHMEM and GASnet for use by global address space languages such as UPC. Finally, there is a lot more work to be done in optimization and analysis of real applications.

References

1. Brightwell, R., Hudson, T., Riesen, R.: The Portals 3.0 message passing interface revision 1.0. Technical Report SAND99-2959, Sandia National Laboratories (1999)
2. Brightwell, R., Maccabe, A.: Scalability limitations of VIA-based technologies in supporting MPI. In: Proceedings of the Fourth MPI Developer's and User's Conference (2000)
3. Blumrich, M.A., Li, K., Alpert, R., Dubnicki, C., Felten, E.W., Sandberg, J.: Virtual memory mapped network interface for the SHRIMP multicomputer. In: ISCA (1994)
4. Druschel, P.: Operating system support for high-speed networking. Communications of the ACM 39(9) (1996)
5. Chen, Y., Bilas, A., Damianakis, S.N., Dubnicki, C., Li, K.: UTLB: A mechanism for address translation on network interfaces. In: ASPLOS (1998)
6. Schaelicke, L.: Architectural Support For User-Level Input/Output. PhD thesis, University of Utah (2001)
7. Araki, S., Bilas, A., Dubnicki, C., Edler, J., Konishi, K., Philbin, J.: User-space communication: A quantitative study. In: SC98: High Performance Networking and Computing (1998)
8. Riddoch, D., Pope, S., Roberts, D., Mapp, G.E., Clarke, D., Ingram, D., Mansley, K., Hopper, A.: Tripwire: A synchronisation primitive for virtual memory mapped communication. Journal of Interconnection Networks 2(3) (2001)
9. Schoinas, I., Hill, M.D.: Address translation mechanisms in network interfaces. In: HPCA (1998)
10. Liu, J., Wu, J., Kini, S., Wyckoff, P., Panda, D.: High performance RDMA-based MPI implementation over InfiniBand. In: SC (2003)
11. Liu, J., Jiang, W., Wyckoff, P., Panda, D., Ashton, D., Buntinas, D., Gropp, W., Toonen, B.: Design and implementation of MPICH2 over InfiniBand with RDMA support. In: IPDPS (2004)
12. Weikuan, Y., Woodall, T.S., Graham, R.L., Panda, D.K.: Design and implementation of Open MPI over Quadrics/Elan4. In: IPDPS (2005)
13. Welsh, M., Basu, A., von Eicken, T.: Incorporating memory management into user-level network interfaces. Technical Report Cornell TR97-1620, Cornell (1997)
14. Harbaugh, L.G.: Reviews: Pathscale InfiniPath interconnect. Linux J. 149 (2006)
15. Binkert, N.L., Saidi, A.G., Reinhardt, S.K.: Integrated network interfaces for high-bandwidth TCP/IP. SIGARCH Comput. Archit. News 34(5) (2006)
16. Bell, C., Bonachea, D.: A new DMA registration strategy for pinning-based high performance networks. In: CAC (2003)
17. Tezuka, H., O'Carroll, F., Hori, A., Ishikawa, Y.: Pin-down cache: A virtual memory management technique for zero-copy communication. In: IPPS/SPDP (1998)
18. Shipman, G.M., Woodall, T.S., Bosilca, G., Graham, R.L., Maccabe, A.B.: High performance RDMA protocols in HPC. In: EuroPVM/MPI (2006)
19. Magoutis, K.: The optimistic direct access file system: Design and network interface support. In: Workshop on Novel Uses of System Area Networks (SAN-1) (2002)
20. Magoutis, K., Addetia, S., Fedorova, A., Seltzer, M.I.: Making the most out of direct-access network attached storage. In: FAST (2003)
21. Magoutis, K.: Design and implementation of a direct access file system (DAFS) kernel server for FreeBSD. In: BSDCon (2002)
22. Reilly, M., Stewart, L.C., Leonard, J., Gingold, D.: Sicortex technical summary (2006), Available http://www.sicortex.com/whitepapers/sicortex-tech_summary.pdf

23. Stewart, L.C., Gingold, D.: A new generation of cluster interconnect (2006), Available http://www.sicortex.com/whitepapers/sicortex-cluster_interconnect.pdf
24. Dally, W.J., Towles, B.P.: Principles and Practices of Interconnection Networks. Morgan Kaufmann, San Francisco (2003)
25. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In: Rolim, J.D.P. (ed.) Parallel and Distributed Processing. LNCS, vol. 1586, Springer, Heidelberg (1999)
26. Scott, S.L.: Synchronization and communication in the T3E multiprocessor. In: ASPLOS (1996)
27. Doerfler, D., Brightwell, R.: Measuring MPI send and receive overhead and application availability in high performance network interfaces. In: EuroPVM/MPI (2006)

Revealing the Performance of MPI RMA Implementations

William D. Gropp and Rajeev Thakur

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{gropp,thakur}@mcs.anl.gov

Abstract. The MPI remote-memory access (RMA) operations provide a different programming model from the regular MPI-1 point-to-point operations. This model is particularly appropriate for cases where there are multiple communication events for each synchronization and where the target memory locations are known by the source processes. In this paper, we describe a benchmark designed to illustrate the performance of RMA with multiple RMA operations for each synchronization, as compared with point-to-point communication. We measured the performance of this benchmark on several platforms (SGI Altix, Sun Fire, IBM SMP, Linux cluster) and MPI implementations (SGI, Sun, IBM, MPICH2, Open MPI). We also investigated the effectiveness of the various optimization options specified by the MPI standard. Our results show that MPI RMA can provide substantially higher performance than point-to-point communication on some platforms, such as SGI Altix and Sun Fire. The results also show that many opportunities still exist for performance improvements in the implementation of MPI RMA.

1 Introduction

MPI-2 added remote-memory access (RMA) operations to the MPI standard. These one-sided operations offer the promise of improved performance for applications, yet users are uncertain whether these operations offer any advantage in most implementations.

A key feature of the one-sided operations is that data transfer and synchronization are separated. This allows multiple transfers to use a single synchronization operation, thus reducing the total overhead. RMA differs from the two-sided or point-to-point model, where each message combines both the transfer and the synchronization. Because of this feature, a performance benefit is most likely to be observed when there are multiple, relatively short data transfers for each communication step in an application.

In this paper, we present a benchmark designed to test such a communication pattern. The benchmark is based on the common "halo exchange" (or ghost-cell exchange) operation in applications that approximate the solution to partial differential equations. We compare a number of implementations of this benchmark

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 272–280, 2007.

with all three MPI RMA synchronization mechanisms: `MPI_Win_fence`, the scalable synchronization (post/start/complete/wait), and passive target (`MPI_Win_lock/unlock`). For each of these mechanisms, MPI defines various parameters (assert options) that may be used by the programmer to help the MPI implementation optimize the operation. In addition, careful implementation can further improve performance [8].

Related Work. A number of papers have explored the performance of MPI RMA. The results in [4] focused on bandwidth for large messages and active-target synchronization in a variant of the ping-pong benchmark, though Table 1 there presents times for a single 4-byte message. The SKaMPI benchmark now supports tests of the MPI one-sided routines [1] and mentions a test similar to our halo test, but without considering varying numbers of neighbors or providing results. The MPI Benchmark Program Library [10] was developed to test the performance of MPI on the Earth Simulator and showed that MPI RMA was faster than the point-to-point operations on that system.

Evaluating the performance of MPI RMA requires careful attention to the semantics of the MPI RMA routines. The broadcast algorithms used in Appendix B and C of [6], for example, rely on `MPI_Get` being a blocking function, which it need not be. In implementations that take advantage of the nonblocking nature of `MPI_Get` allowed by the MPI Standard (for example, MPICH2 [8]), the code in Appendix B and C of [6] will indeed go into an infinite loop.

Papers that discuss the implementation of MPI RMA naturally include performance measurements; for example, see [2,9]. The test we use in this paper is similar to Wallcraft's halo benchmark [11], but that benchmark does not use MPI one-sided communication and uses only four neighbors in the halo exchange. Wallcraft's halo benchmark has also been used in comparing MPI with other programming models [3].

2 The Benchmark

Our benchmark exchanges data with a selected number of partner processes. It mimics a halo, or ghost-cell, exchange that is a common component of parallel codes that solve partial differential equations. The code for this pattern, using MPI point-to-point communication, is as follows:

```
for (j=0; j<n_partners; j++) {
    MPI_Irecv( rbuffer[j], len, MPI_BYTE, partners[j], 0,
               MPI_COMM_WORLD, &req[j] );
    MPI_Isend( sbuffer[j], len, MPI_BYTE, partners[j], 0,
               MPI_COMM_WORLD, &req[n_partners+j] );
}
MPI_Waitall( 2*n_partners, req, MPI_STATUSES_IGNORE );
```

In the case of two partners, the neighbor processes are the processes with ranks one greater and one less than the rank of the process. In the case of four partners,

the neighbor processes mimic a two-dimensional decomposition. In the case of eight partners, the eight neighbors in a two-dimensional decomposition are used.

This test is chosen because it allows us to separate the data transfers (in the RMA case, the `MPI_Put` and `MPI_Get` calls) from the synchronization (e.g., the `MPI_Win_fence` call). It also better reflects the communication in many simulation applications than does the standard ping-pong test.

Each of the MPI RMA methods allows different options for optimization. For example, there are various "assert" options for `MPI_Win_fence`. Our test allows the selection of the following options.

Fence. This is active-target synchronization with `MPI_Win_fence`.

1. Allocate send and receive buffers with `MPI_Alloc_mem`.
2. Specify "no_locks" in `MPI_Win_create`.
3. Provide assert option `MPI_MODE_NOPRECEDE` on `MPI_Win_fence` before RMA calls and all of `MPI_MODE_NOSTORE`, `MPI_MODE_NOPUT`, and `MPI_MODE_NOSUCCEED` on the `MPI_Win_fence` after the RMA calls.

Post/Start/Complete/Wait. This is scalable active-target synchronization with `MPI_Win_post`, `MPI_Win_start`, `MPI_Win_complete`, and `MPI_Win_wait`.

1. Allocate send and receive buffers with `MPI_Alloc_mem`.
2. Specify "no_locks" in `MPI_Win_create`.

Passive. This is passive-target synchronization with `MPI_Win_lock` and `MPI_Win_unlock`.

1. Use locktype `MPI_LOCK_SHARED` (instead of `MPI_LOCK_EXCLUSIVE`).
2. Do not use a separate `MPI_Barrier` for the target processes to know that all RMA operations have completed. This is relevant for applications that may have an algorithmic reason for knowing that RMA operations are complete, such as a required `MPI_Allreduce`.

Since an implementation may require that only memory allocated with `MPI_Alloc_mem` be used for passive-target RMA, we do not attempt to use the passive-target mode without allocating memory in this way.

The testing methodology is the same as that used in `mpptest` and was described in [5]. It uses the minimum of an average time, where the time of an individual test (containing multiple iterations of the basic communication test) is large relative to the granularity and precision of the clock. Since the tests are implemented within the `mpptest` code, all of the many options for controlling message sizes and measurement details are available. The tests are available as part of the current distribution of `mpptest`, available at [7]. A script, `runhalo`, is provided that runs the tests with the various RMA optimization options.

To better understand the tests, we also measured the halo exchange as implemented with `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall` (as in the example code) and with persistent sends and receives. In addition, since `MPI_Win_fence` can be implemented with `MPI_Barrier` on cache-coherent SMPs where immediate direct-memory copy is used for the MPI RMA operations, we also measured the performance of `MPI_Barrier`.

3 Results

We ran our tests on a variety of platforms and with a variety of MPI implementations. Results for the native (vendor-supplied) implementations are provided for SGI Altix, Sun Fire, and IBM p655+ SMPs. We also include results for MPICH2 version 1.0.5 and Open MPI 1.2.0 on a Linux cluster.

The results of testing the performance-optimization features showed that only a few optimizations are exploited by the implementations we tested. Table 1 summarizes which optimization approaches provided a significant, measurable benefit in our tests. In the discussion of each platform, the results with the best choice of options are used.

Table 1. Optimizations that were observed to help in the halo tests. An "X" appears in the "All" row only if using multiple optimizations provides an improvement over a single optimization. "NA" means that the MPI implementation does not support that feature. No options provided a benefit on the IBM p655+.

Option	SGI Altix	SUN Fire	IBM p655+	MPICH2	Open MPI
AllocMem w Fence		X			
Nolocks w Fence					
Asserts w Fence				X	X
All w Fence					
AllocMem w PSCW	NA	X			
Nolocks w PSCW	NA				
All w PSCW	NA				
Shared locks	X	X			

The rest of this section describes the performance of the different RMA synchronization modes using the best set of optimization values. In the interests of space, we provide graphs for only a subset of our results, summarizing the measurements in the text.

3.1 SGI Altix

We ran our tests on three different SGI Altix SMP systems that are part of the Columbia supercomputer at the NASA Ames Research Center. These were the single-core SGI Altix 3700 and Altix 3700 Bx2 and the dual-core Altix 4700; the results in this paper are from the Altix 3700 Bx2. SGI's MPI implementation does not support the post/start/complete/wait method of synchronization, only fence and lock-unlock.

Figure 1 shows that the Altix has excellent RMA performance. Lock-put-unlock without an additional barrier performs significantly better than any other form of communication. For the 8-neighbors case, it is ten times faster than send-receive. Even the fence method for 8 neighbors (put-8) is more than twice as fast as send-receive.

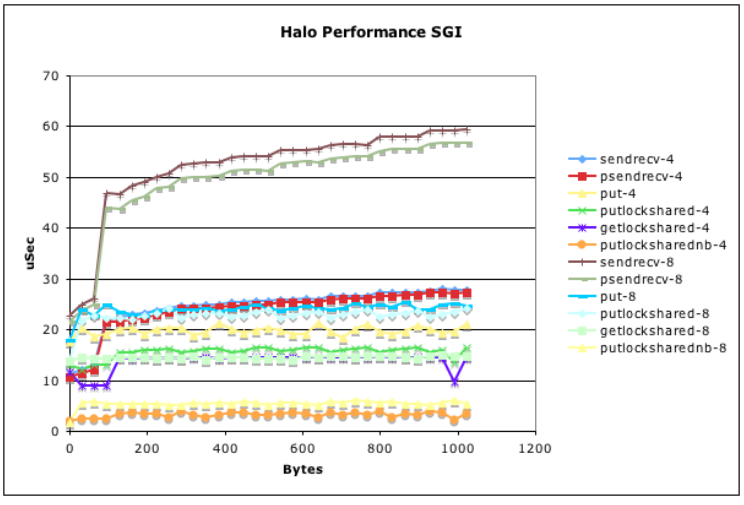


Fig. 1. Performance of halo exchange on SGI Altix with 16 processes. The best RMA results are compared with point-to-point; the legend indicates the number of neighbors (e.g., put-4 is put/fence with four neighbors, psendrecv-8 is persistent send/receive with eight neighbors, and nb stands for no barrier).

A surprising aspect of the Altix results is that the RMA optimization features in the MPI calls (e.g., the assert values in `MPI_Win_fence`) have no measurable effect, nor does using memory allocated with `MPI_Alloc_mem` (for fence). While this is attractive for the user (nothing to do), a closer look at all the data we collected suggests that additional optimizations could help in some cases. For example, in the two-neighbor case, put-fence was slower than send-recv by 50%. But since lock-unlock was significantly faster, a tuned version of fence that takes advantage of user-provided asserts should also be able to outperform send-recv.

3.2 Sun Fire

We ran our tests on the Sun Fire SMP cluster at the RWTH Aachen University using Sun's MPI. The specific machine we ran on was a Sun Fire E2900 with eight dual-core UltraSPARC IV 1.2 GHz CPUs. Figure 2 shows a subset of the results. As on the Altix, the performance of lock-unlock without an additional barrier is the best of all communication methods—it is twice as fast as send-receive. The performance of MPI RMA on this system is quite good if the memory used is allocated with `MPI_Alloc_mem`. The other optimization options had little or no effect on the performance of the halo tests. In particular, the `MPI_Win_fence` options had no effect. One unusual feature of this implementation is the extraordinarily long time required by `MPI_Alloc_mem` and `MPI_Win_create`. Times of several seconds were measured; we rarely saw these routines take less than a few seconds when using 16 processes.[1]

[1] We were told that the performance problem with `MPI_Alloc_mem` has been fixed in Sun's ClusterTools 7; the version on the machine was ClusterTools 5.

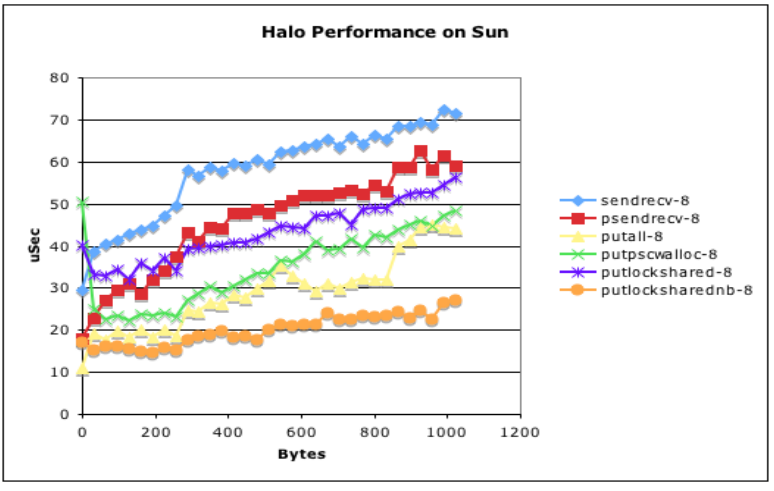


Fig. 2. Performance of 8-neighbor halo exchange on Sun Fire SMP with 16 processes in `MPI_COMM_WORLD`. putpscwalloc is the scalable synchronization with `MPI_Alloc_mem`. putlockshared is passive target with shared locks, and putlocksharednb omits the barrier that is necessary to ensure completion at the target.

3.3 IBM p655+

We ran our tests on the DataStar machine at the San Diego Supercomputer Center with IBM's MPI. The specific node we ran on was an IBM p655+ 8-way SMP. The p655+ has 1.7 GHz POWER4+ CPUs. Nodes in DataStar are connected with the Federation Switch; however, as our tests used a single node, the switch was not used.

With eight processes on an eight-node SMP, the RMA performance was very poor, on the order of forty times slower than the point-to-point performance. With seven processes on the same eight-node SMP, the RMA performance was still poor but an order of magnitude faster than with eight processes. This case is shown in Figure 3. The significant change in performance between eight and seven processes suggests that a thread is used for implementing the RMA operations and that the implementation is not prepared to handle the case where there are more threads than processors. To test this hypothesis, we also ran with four MPI processes on an eight-processor system. The performance in that case was further improved over the seven-process case but was still poor relative to the point-to-point version. An `MPI_Barrier` on this system takes roughly 9 μsec on 8 processes, so the cost of a barrier or barrier-like synchronization is not a major contributor to the high cost of RMA on this system.

3.4 Linux Cluster

We also ran the tests on the Jazz cluster at Argonne, which has 2.4 GHz Pentium Xeon nodes and both a Myrinet 2000 and 100 Mb/s Ethernet interconnect. We

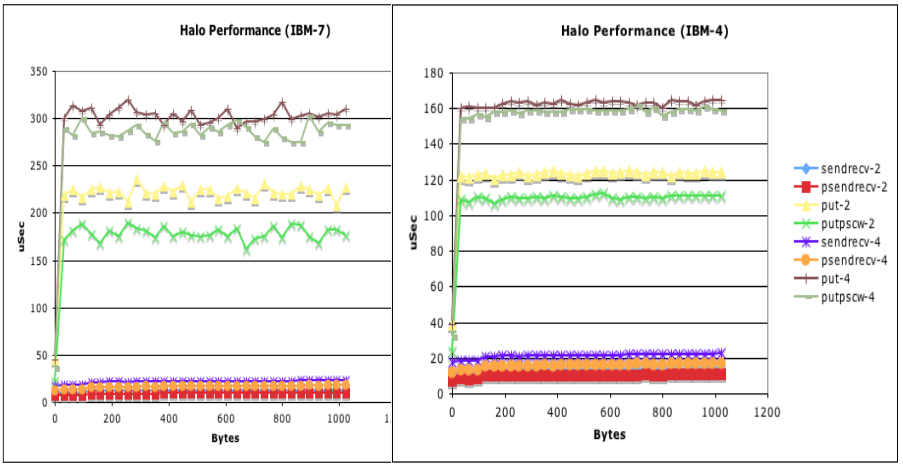


Fig. 3. Performance of RMA on IBM p655+. The chart to the left is with 7 processes on an 8-node SMP; to the right is 4 processes on an 8-node SMP. Results for two and four neighbors are shown using the two active target synchronization methods (put and putpscw in the legend) and point to point with nonblocking and persistent send/receive (sendrecv and psendrecv in the legend, respectively).

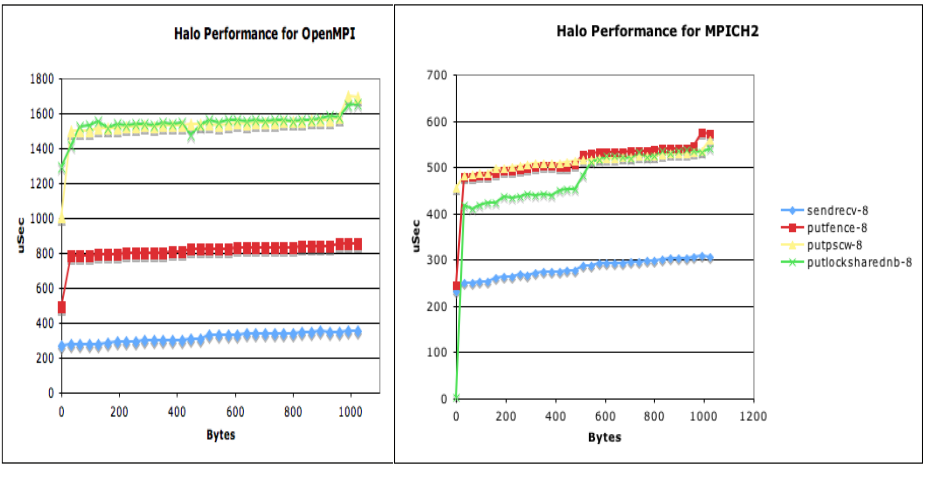


Fig. 4. Performance of 8-neighbor halo exchange with 16 processes on the Linux cluster by using Open MPI (left) and MPICH2 (right)

used two MPI implementations, MPICH2 1.0.5 and Open MPI 1.2.0. The cluster uses an older version of the native GM library for Myrinet, and we could not build Open MPI for that version. Hence we used TCP over Myrinet for communication with both MPICH2 and Open MPI. As the results in Figure 4 show,

the best performance was achieved with the point-to-point operations for both implementations. The reason is that in the absence of hardware and software support for RMA from the network-transport layer, the MPI RMA operations are simply implemented on top of lower-level point-to-point operations. Nonetheless, RMA with MPICH2 performs significantly better than with Open MPI. Some of this performance improvement is due to the optimizations in MPICH2 that minimize the synchronization overhead associated with MPI RMA [8].

4 Conclusions

We have shown that implementations of MPI RMA can provide a performance advantage on systems with hardware support for remote-memory operations, particularly when there are multiple RMA operations per synchronization operation. The SGI Altix and Sun Fire provided surprisingly good performance for the passive-target RMA operations; in fact, the performance was so good that it may be possible to improve the performance of the active-target RMA methods by making use of the approach used for the passive-target RMA.

We measured surprisingly poor performance on an IBM SMP. We suspect that the implementation is not optimized for MPI RMA operations and relies on separate threads that may be running in a polling mode, thus leading to very poor performance when there are fewer processors than at least two times the number of MPI processes.

Few of the flags provided by the MPI standard are exploited by the implementations. This situation was reflected in the surprisingly high overhead for active-target RMA operations on most of the platforms. We hope that our benchmark will encourage MPI implementors to exploit these features.

Acknowledgments

We thank the RWTH Aachen University, NASA Ames, and the San Diego Supercomputer Center for providing computing time on their systems. We particularly thank Subhash Saini and Dale Talcott for running the tests on the Altix machines and Anthony Chan for running the tests on the Linux cluster.

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

1. Augustin, W., Straub, M.-O., Worsch, T.: Benchmarking one-sided communication with SKaMPI 5. In: Di Martino, B., Kranzlmüller, D., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3666, pp. 301–308. Springer, Heidelberg (2005)

2. Booth, S.: Mourão, E.: Single sided MPI implementations for SUN MPI. In: Proceedings of Supercomputing 2000 (CD-ROM), Dallas, TX, November 2000. IEEE and ACM SIGARCH. EPCC, The University of Edinburgh (2000)
3. Co-Array Fortran vs MPI http://www.co-array.org/cafvsmpi.htm
4. Gabriel, E., Fagg, G.E., Dongarra, J.: Evaluating the performance of MPI-2 dynamic communicators and one-sided communication. In: Dongarra, J.J., Laforenza, D., Orlando, S. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2840, pp. 88–97. Springer, Heidelberg (2003)
5. Gropp, W.D., Lusk, E.: Reproducible measurements of MPI performance characteristics. In: Margalef, T., Dongarra, J.J., Luque, E. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1697, pp. 11–18. Springer, Heidelberg (1999)
6. Luecke, G.R., Spanoyannis, S., Kraeva, M.: The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a Cray T3E-600. Concurrency and Computation: Practice and Experience 16(10), 1037–1060 (2004)
7. MPPTEST - Measuring MPI Performance http://www.mcs.anl.gov/mpi/mpptest
8. Thakur, R., Gropp, W., Toonen, B.: Optimizing the synchronization operations in MPI one-sided communication. International Journal of High-Performance Computing Applications 19(2), 119–128 (2005)
9. Träff, J.L., Ritzdorf, H., Hempel, R.: The implementation of MPI-2 one-sided communication for the NEC SX-5. In: Proceedings of Supercomputing' (CD-ROM), Dallas, TX, November 2000. IEEE and ACM SIGARCH NEC Europe Ltd. (2000)
10. Uehara, H., Tamura, M., Yokokawa, M.: An MPI benchmark program library and its application to the Earth Simulator. In: Zima, H.P., Joe, K., Sato, M., Seo, Y., Shimasaki, M. (eds.) ISHPC 2002. LNCS, vol. 2327, pp. 219–230. Springer, Heidelberg (2002)
11. Wallcraft, A.J.: SPMD OpenMP versus MPI for ocean models. Concurrency: Practice and Experience 12(12), 1155–1164 (2000)

Distributed Real-Time Computing with Harness*

Emanuele Di Saverio[1], Marco Cesati[1], Christian Di Biagio[2], Guido Pennella[2], and Christian Engelmann[3]

[1] Department of Computer Science, Systems, and Industrial Engineering, University of Rome "Tor Vergata", Rome, Italy
[2] Applied Research & Technology Department, MBDA Italia SPA, Rome, Italy
[3] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA
emanuele.disaverio@alice.it,cesati@uniroma2.it,
{christian.di-biagio,guido.pennella}@mbda.it,engelmannc@ornl.gov

Abstract. Modern parallel and distributed computing solutions are often built onto a "middleware" software layer providing a higher and common level of service between computational nodes. Harness is an adaptable, plugin-based middleware framework for parallel and distributed computing. This paper reports recent research and development results of using Harness for real-time distributed computing applications in the context of an industrial environment with the needs to perform several safety critical tasks. The presented work exploits the modular architecture of Harness in conjunction with a lightweight threaded implementation to resolve several real-time issues by adding three new Harness plug-ins to provide a prioritized lightweight execution environment, low latency communication facilities, and local timestamped event logging.

Keywords: Distributed Computing, Middleware, Real-Time, Harness, Plugin.

1 Introduction

Parallel and distributed computing solutions provide the means for computational performance for High-End Computing (HEC) applications beyond the limits of single processor technology. The actual implementation of complex parallel and distributed software systems can be enormously eased by the adoption of an intermediate software layer, a "middleware". A middleware is defined as "...a connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network." [1]. A very specific topic of HEC applications is real-time computation.

* The research at Oak Ridge National Laboratory (ORNL) is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. ORNL is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 281–288, 2007.

The term *real-time* pertains to computer applications whose correctness depends not only on results, but also on the time at which results are delivered. A *real-time system* (RTS) is a computer system that is able to run real-time applications and fulfill their requirements in a deterministic fashion. Thus, when defining a real-time system, we actually define requirements about its response time, meaning with this average value or the tail distribution of it. Distributed real-time systems development requires a well suited real-time oriented middleware as a supporting layer.

2 Previous Work

Traditional solutions in distributed real-time environments refer to Parallel Virtual Machine (PVM) [2] and Message Passing Interface (MPI) [3] libraries. Although PVM is a solid and simple solution, its process-based architecture is a little outdated, and the service set does not fit well into an industrial context. MPI is not well suited when compared to modern middlewares because it provides just a communication abstraction. More recent and rich products include Real-Time Innovations Data Distribution Service [4] (RTI DDS, formerly NDDS) and the Adaptive Communication Environment (ACE) Object Request Broker (ORB) in conjunction with TAO [5]. RTI DDS provides a communication abstraction data-centric layer that realizes a publish-subscribe semantic. It is a performing and well-featured product, which offers real-time oriented features, like a fine tunable QoS performance level, and an efficient low-latency implementation based on an open standard from OMG group [6]. However, it is especially suited for dynamically changing network topologies and to cover reliability issues. Moreover, it is a commercial (and thus closed) product, while in the industrial context being able to lower costs and customize the product at will is of key importance. The ACE ORB, on the other hand, is an open source project that implements the Object Request Broker semantic. TAO is a very complex and complete middleware, but it is designed around the ORB specification and is meant to be used with the existing plethora of CORBA services. This means that the internal architecture of TAO involves many different components. Moreover, its service-oriented nature makes it not easily tailorable for embedded applications. The approach described in this paper is more simple and streamlined, it avoids the role of the broker node, and it integrates nicely with the Harness framework.

3 Modern Middleware: Harness

Our effort focuses on an emerging technology in this field, the Harness project, a joint development effort between the Oak Ridge National Laboratory (ORNL), the University of Tennessee, and Emory University. Harness is a distributed, reconfigurable and heterogeneous computing environment that supports dynamically adaptable parallel and distributed applications. The unique feature of Harness relies on its almost total level of pluggability. The aim is to build a virtual

environment that can dynamically change (almost) anything at runtime. In this highly adaptable framework, several parallel and distributed user applications can reside, all executed on top of its distributed virtual machine (DVM) and runtime environment (RTE) concepts, a PVM successor. Harness runs a RTE on every computational unit as the "shell" in which it hosts the user applications and the resource management routines that belong to the distributed environment. Every RTE is realized by a Harness kernel, the core of the unit, which is capable of loading and unloading plugin modules and consists of a communication module, a module dedicated to process control, and a module dedicated to resource management, plus possibly a number of other plugins.

Several Harness prototypes have been developed, in Java and C. The work presented in this paper focuses on the C variant developed at ORNL [7], which runs on GNU/Linux. Its design focuses on a lightweight and pluggable middleware layer with a Harness kernel running as a Linux daemon process. The kernel performs process management, thread pool management, and dynamic plugin loading/unloading. The process management module can fork and execute a user application, and provides means for passing arguments, sending input, and retrieving output for these external processes. The thread pool is the heart of the lightweight execution environment provided by Harness. It creates a set of working threads that keep trying to empty a job queue data structure. It provides interfaces for adding a new job to the queue with proper arguments and cleanup function in case of thread cancellation. The plugin loader builds on top of the Linux dynamic library loader, and provides interfaces for loading/unloading modules and publishing their functionalities to the whole runtime environment. The communication facilities are effectively provided through RMIX (Remote Method Interface eXtensible) [8]. RMIX is a dynamic, heterogeneous, reconfigurable communication framework that allows software components to communicate using various RMI/RPC protocols by employing provider plugins in order to support different protocol stacks. RMIX emulates the Java RMI structure, allowing components to remotely call methods, retrieve the output, and export locally available methods.

4 Existing Real-Time Issues

Within the middleware definition stated in Section 1, we can outline a set of requirements expected from such a software. The first assumption we make is that the services on top of which the middleware is built retain themselves real-time capabilities. We cannot avoid this assumption because the aim is to work on a middleware that is a relatively high-level software, thus built on top of a set of services, and the performances of the resulting system are bound to those of that services. Secondly, remote services have to be designed without concurrency issues. In this context, the meaning of concurrency is twofold:

– **Call Concurrency:** simultaneous access of the same service offered by the same computational resource

– **Service Concurrency:** simultaneous access of different services offered by the same computational resource

A real-time middleware has to guarantee both kinds of concurrency support in order to be seamlessly scalable with the growth of the execution flows. Finally, services have to be locally executed without scheduling issues by providing a proper priority-aware execution environment. The hosted application should be able to run a number of real-time tasks that are expected to be scheduled in a deterministic manner. In order to address all these issues, three Harness plugins have been developed:

– A plugin to provide a prioritized lightweight execution environment
– A plugin for low latency communication facilities
– A plugin to support local timestamped event logging

5 Developed Plugins

The aim of our work is exploiting the pluggable nature of Harness in order to implement a set of services enabling the development and execution of successful real-time applications.

5.1 Real-Time Thread Pools

The lightweight execution environment provided by Harness is designed for great efficiency, but lacks in direct support for operating system scheduler directives. The thread pool solution is indeed a lightweight solution for job processing, but lacks the ability to exploit the preemptability of the latest Linux kernels. POSIX threads, like processes, can control their scheduling priority and contention scope, which can be set to either process or system scope. By using the latter it is possible to grant absolute schedule priority to the thread. Therefore, the first developed plugin focuses on providing a greater level of control on thread pools to user applications. It allows to define an arbitrary number of job classes, and for each level to specify the scheduler policy and priority of the related thread pool. Each pool tries to empty a different job queue. If not otherwise specified, the plugin is configured by default to create three pools of threads, in addition to the original one, it adds two pools entirely made of threads with real-time scheduling properties.

The first additional thread pool adopts a round-robin scheduling algorithm with priority p_1. The second additional thread pool uses a first-in/first-out scheduling algorithm with priority p_2, where $p_1 < p_2$ and global contention scope hold. This way we scaled the priority of the executing threads from one to three levels, allowing the application executing on top of Harness to have control over the scheduling of its tasks. The default configuration should provide more than enough means for executing real-time applications without worries for scheduling issues. If a more fine-grained solution is needed, it is still possible to explicitly specify the pool parameters.

5.2 Real-Time Remote Procedure Calls

The second plugin faces a problem of the RMIX framework in its actual form. The standard provider plugin builds on top of the TCP transport layer. This solution, while providing a reliable and stream oriented communication, is not well suited for distributed real-time applications. In order to address this issue, an RMIX provider plugin was built to implement the RMIX primitives over UDP, while keeping a real-time job (as explained in the previous subsection) serving the incoming and outgoing communications. This way we bring into the Harness middleware layer a more efficient implementation of the Remote Procedure Call communication scheme built onto a connectionless transport level. While losing the reliability and the complex acknowledgment system of TCP, we gain performance in response time metric, both in its absolute value and in variance of its distribution. This solution exploits the pluggable nature of the RMIX Framework, that itself is seen as a plugin by the Harness Framework. This double-layered pluggability has the added benefit of not requiring modifications to applications already using the RMIX Communication Framework.

5.3 Real-Time Event Logging

A common source of unpredictable latency is the access to file or screen I/O devices, as a plain `printf()` function call is a very time consuming task. While this effect can easily be ignored in standard distributed applications, it cannot be tolerated in a distributed real-time system. The third developed plugin implements a simple event logging system. Loading this module enables the application to push the event to be logged in a temporary shared buffer, while storing information about the source of the event, the timestamp, and the description of the event itself. This operation is a low overhead one, while the buffer will be emptied by a regular Harness thread waiting on the event queue, and optionally formatting the output in a simple XML or plain text file. This way an application can effectively maintain logs of events with accurate timestamps in a lightweight fashion, that is, without perturbating the execution environment as seen by real-time tasks.

6 Experimental Tests and Evaluation

The development process in industrial, high performance, and time-critical environments includes an extensive and thorough performance testing. It is important to build a test environment that resembles the operational environment as closely as possible, both in hardware and software, and to perform tests in adequately set up stress conditions.

6.1 Test Environment

The Operational Environment of an industrial time-critical application is mainly composed by “Command and Control” (C2) applications. We model a C2

distributed application as constituted by components of one of three types [9]: a *sensor* type component that receives the data from the environment, an *elaborator* component that computes the actions to be taken in response to data received from sensor, and an *actuator* component that finally executes these actions in order to modify one or more entities of the environment to be controlled. Following this scheme, a distributed application was developed in order to realistically mimic this behavior. The application testing utilizes a fictional remote control of a vector in a 2D space along the two coordinates and speed. Such an application is computationally very similar to real-world C2 applications used in industrial, aerospace, and military contexts, because it involves geometric algorithms on 2D polygons as well as trigonometric and floating-point operations. Moreover, to perform stress tests, benchmark programs are used to synthetically generate the load that simulates extreme operational conditions. Stress tests are necessary for an industrial and mission-critical real-time system to check if the software retains its performance level even in presence of high or spike load. The *Ubench* and *Hackbench* benchmarks were used in order to study the behavior of the application under varying load circumstances.

6.2 Test Results

The test were performed with unloaded systems, with Ubench load (CPU and memory), and with different Hackbench load parameters. The goal was to determine how the performance of the distributed application as a whole was affected in scenarios of high load. Figure 1 shows the round trip time (RTT) of the distributed application in the Ubench-loaded configuration, *i.e.*, the time that occurs between the data capture from the *sensor* component and the *action* taken by the *actuator* component. The total RTT of the application roughly doubles under loaded conditions, due to the number of context switches needed between the benchmark program and the application, but never exceeds the value of 2.2 milliseconds, as reported in Table 1.

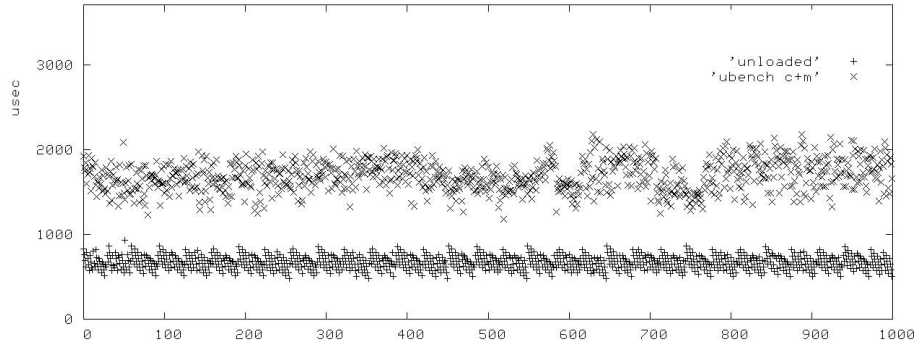


Fig. 1. Computational and Memory Load Sensitivity - Ubench

Table 1. Round Trip Time Latency Comparison - Ubench

	Average	Standard Deviation	Maximum
Unloaded	669.67μs	81.62μs	932μs
Loaded	1694.39μs	178.45μs	2182μs

In the Hackbench tests we measured mean, maximum, and standard deviation values of the application RTT while varying the coefficient of load (passed as parameter to the benchmark). The results (see Figure 2) do not show a significant relation between the rise of the load parameter and the RTT either in its maximum value or in mean and variance. It is worth noting that both load configurations were tested thoroughly with increasing loads until reaching instability of the host systems. Yet the performance of the distributed application was predictable and reliable, and retained real-time class performance.

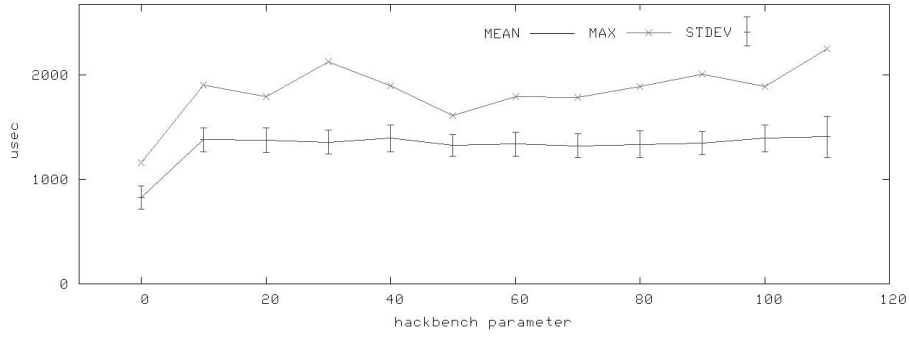


Fig. 2. Task Scheduling Load Sensitivity - Hackbench

7 Conclusions and Future Work

In this paper, we described recent research and development efforts in building an open source runtime and communication middleware layer for distributed real-time applications. Within this context, Harness represents an optimal choice of *base line* due to the extreme dynamic modularity its pluggable architecture offers. We exploited this feature to build the desired set of real-time functionalities in the form of *plugins* that realize communication and priority-aware execution services with a real-time level of performance.

The performed tests show how a distributed real-time application (in this case, a Command & Control application) can utilize the developed features. Our work, however, makes the assumption that the underlying software layers can provide an adequate level of performance. This is generally not true in the chosen test environment.

Future work in this area will include porting the Harness middleware layer onto a dedicated *hard real-time* operating system and network stack. Xenomai

seems to be the best candidate of the set of real-time OS's, because it can provide access to (hard) real-time features, while keeping an external POSIX interface (Xenomai Skin Technology [10]). As an added bonus, Xenomai offers a complete real-time networking stack with its integrated RTNET technology. Another solution consists of adopting a completely different network technology that offers an entire stack of real-time-oriented features, like Infiniband and its related protocols. Ongoing research activities are conducted in this direction by the Applied Research & Technology Department of MBDA Italia S.p.a.

References

1. Bray, M.: Middleware, Software Technology Review at Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA(1997), Available at http://www.sei.cmu.edu/str/descriptions/middleware.html
2. Geist, G.A., Beguelin, A., Dongarra, J.J., Jiang, W., Manchek, R., Sunderam, V.S.: PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, USA (1994)
3. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. MIT Press, Cambridge, MA, USA (1996)
4. Real-Time Innovations, Inc. Santa Clara, CA, USA: Data Distribution Service. (2007), Available at http://www.rti.com/products/data_distribution/
5. Washington University, St. Louis, MO, USA: Adaptive Communication Environment (ACE) with TAO (2007), Available at http://www.cs.wustl.edu/ schmidt/ TAO.html
6. Object Management Group, Inc. Needham, MA, USA: Data Distribution Service for Real-time Systems (2007), Available at http://www.omg.org/technology/ documents/ formal/data_distribution.htm
7. Engelmann, C., Geist, G.A.: A lightweight kernel for the harness metacomputing framework. In: Proceedings of the 14^{th} Heterogeneous Computing Workshop (HCW) 2005, in conjunction with the 19^{th} International Parallel and Distributed Processing Symposium (IPDPS), Denver, CO, USA (2005)
8. Engelmann, C., Geist, G.A.: RMIX: A dynamic, heterogeneous, reconfigurable communication framework. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 573–580. Springer, Heidelberg (2006)
9. Ravindran, B.: Engineering dynamic real-time distributed systems: Architecture, system description language, and middleware. IEEE Transactions on Software Engineering 28, 30–57 (2002)
10. Gerum, P.: Xenomai - Implementing a RTOS emulation framework on GNU/Linux (2004), Available at http://download.gna.org/rtai/documentation/vesuvio/ html/xenomai

Frequent Itemset Minning with Trie Data Structure and Parallel Execution with PVM

Levent Guner and Pinar Senkul

Department of Computer Engineering, Middle East Technical University,
Ankara, Turkey
{Leventguner,senkul}@ceng.metu.edu.tr

Abstract. Apriori algorithm is one of the basic algorithms introduced to solve the problem of frequent itemset mining (FIM). Since there is a new generation of affordable computers with parallel processing capability and it is easier to set up computer clusters, we can develop more efficient parallel FIM algorithms for these new systems. This paper investigates the use of trie data structure in parallel execution of Apriori algorithm, the potential problems during implementation, performance comparison of several parallel implementations and in order to increase the efficiency, proposes a new way of message passing for parallel Apriori on a computer cluster with PVM.

Keywords: Apriori, PVM, Parallel Execution, Trie, Message Passing.

1 Introduction

Frequent itemset mining (FIM) is a relatively new field of data mining. It is a part of associative rule mining where sample data is given as a set of itemsets and the aim is to find associations among itemsets. In FIM, the aim is to find sets of items that occur over a predefined frequency (i.e. support threshold) among all data, which is a set of itemsets, where basket is a common word to describe each itemset [1]. FIM is a necessary step in association rule mining, and used in different applications such as gene analysis and customer relationship management.

Apriori is one of the basic algorithms used for frequent itemset mining. It is a breadth first search algorithm that generates the frequent itemsets in a bottom-up fashion. Candidates of size k are generated from frequent itemsets of sizes k-1. Then the candidates that contain infrequent k-1 subsets are removed before the support count, because any subset of a frequent itemset has to be frequent. Hence the name 'Apriori' is given for this property [2].

After pruning set of candidates, the database is scanned for the actual support count. If a candidate does not meet the minimum support frequency, it is eliminated (i.e. not found in at least the minimal number of transactions in the database).

In the search for faster techniques, some other well known FIM algorithms have been developed like FP-growth [3] and Eclat [4]. There has also been an interest in different data structures and search techniques used with these basic algorithms.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 289–296, 2007.

Selection of data structure and search technique is important for the efficiency and scalability of the FIM algorithms.

In this paper parallel scalability of the Apriori algorithm is investigated while some data structure related techniques are evaluated for their contribution to the overall performance.

The paper is organized as follows: In section two, brief information of previous work on the subject is given. In section three, our implementation, with explanation of data structures and speed-up techniques, and parallelization efforts of Apriori with PVM is explained. Experiments and their results are shown in fourth section. Conclusions and future works are discussed in the last section.

2 Previous Work

Apriori algorithm was developed by Agrawal and Srikant [2]. In its original form hash trees were used as the data structure. Bodon has conducted an extensive survey on frequent itemset mining, and proposed the trie data structure as simple and efficient to be used in Apriori. He has also offered some interesting methods used to accelerate Apriori [5-7]. These are filtering transactions, frequency ordering of the candidate trie, early breakup in the support count function, simultaneous traversal of the trie, binary and reverse binary search in the candidate tree.

As a result, he has produced a very competitive implementation with the combination of these new techniques. Some of these techniques are explained in the next section while discussing our own implementation.

Several alternative parallel execution methods of Apriori algorithm were evaluated in a comprehensive paper by Han, Karypis and Kumar [8]. While it is possible to execute Apriori in parallel with several different methods, the count distribution method mentioned in their paper and in Agrawal's paper [9] is the most efficient and a similar method will form the basis of our implementation.

A simple parallel Apriori implementation was also offered by Ye [10]. Ye has made some experiments on Apriori with trie on a parallel computer; however, details of the implementation are not fully disclosed in his paper.

3 Implementation

3.1 Data Structure

Trie, though a simple data structure, can be implemented in several ways. In our implementation trie was constructed of several binary trees in each level. It is possible to use arrays or hash tables for this purpose; however, since the implementation language is C++, *binary trees* provided by STL makes the implementation fairly simpler. The balanced binary tree is represented with *map* in STL. It offers an automatic sorting property and built-in binary searches. The associative array property of the map allows the storage of support count values and the storage of information for the connected edges that are represented as pointers to the maps in the next level.

Transaction database is represented as a binary tree, but in this case the associative property allows the storage of multiple occurrences of the same transaction in one slot.

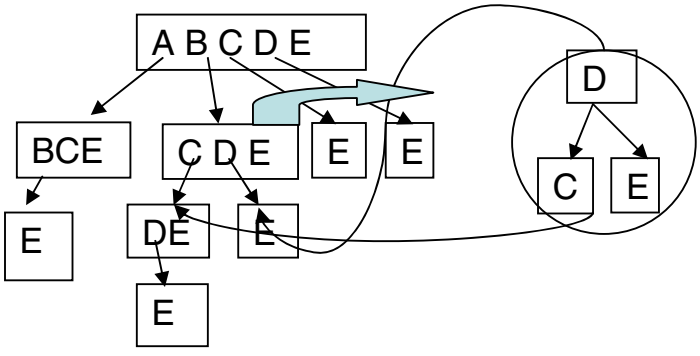


Fig. 1. Sample Trie Structure from our implementation

3.2 Parallel Execution of Apriori

In this work, it is aimed to evaluate the Apriori algorithm's scalability on a parallel computer, while using some speed-up techniques proposed by Bodon [5]. The system used for parallel execution is a computer cluster with PVM as the message passing interface and each computer has an AMD Athlon 1700+ CPU and 256MB's of memory.

The support count is the computationally most expensive step in Apriori; therefore it is advantageous to start parallel processing from the point of support calculation. The basic approach is to distribute the transaction database into several processors to divide the support count step.

The basic and the least efficient method is using string-matching between candidates and transactions. In the implementation of this work, in order to store candidate itemsets, a trie is used (Figure 1). The circle shows the STL *map* containing C, D, and E, in the level 2. Curved arrows show the pointers to the maps in the next level in the trie. In this trie, there are eight 2-itemsets, {A,B}, {A,C}, {A,E}, {B,C}, {B,D}, {B,E}, {C,E}, {D,E}, four 3-itemsets, {A,B,E}, {B,C,D}, {B,C,E}, {B,D,E}, and one 4-itemset {B,C,D,E}.

We distribute the transaction database to the slave processes in the beginning of the algorithm. As the first step, the one item frequent itemsets are found in the first scan of the database by the master process, then, using them the 2-itemset candidates are formed and sent to the slave processes.

At this point there are several alternative ways of sending the candidates from the candidate trie.

Version 1. The first approach is to extract the candidates and send them as arrays and let the slave processes build tries. This lets us manipulate the candidates according to their orderings, for example according to their frequency as explained in the next section. Our experiments have shown that this approach scales poorly with the number of slave processes. This is shown with blue arrows in Figure 2.

Version 2. The alternative solution is to transfer the trie as in its original form. This way the candidates are not extracted, sent and slave processes reconstruct the trie; but the slave processes listen to the master for the steps to take, as master traverses its own candidate trie while encoding the steps it takes. This approach is shown with

yellow arrows in Figure 2. This solution seemed to scale well but only up to a certain value of candidates generated and sent. The PVM process also has problems like random terminations or extremely long response times.

Version 3. To be able to overcome the problems with previous approaches, in this work, a new solution is developed and a single message for the encoded trie is created. In this approach the slave processes do not have to listen to the master but decode this single message in order to build the candidate trie. This approach is shown with red arrows in Figure 2.

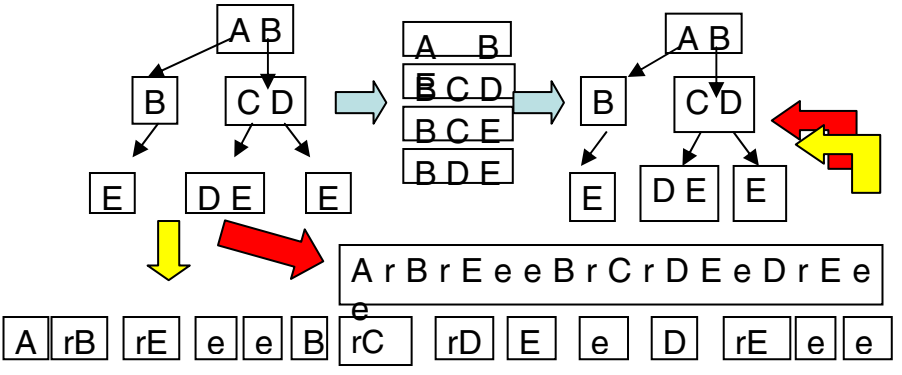


Fig. 2. Trie with 3-itemset candidates derived from the trie in Figure 1 is sent to slave processors with three alternative methods. The letter 'r' signals for a recursion and 'e' signals the end of the recursion call.

For the return of the support count results, after finishing their support count, the slave processes create an array and send it back. Since the candidate tries are exactly the same in each process, master process arithmetically add these arrays and traverse the candidate trie to insert the values. The elimination of candidates with low support count and creation of new candidates are combined in a single function.

3.3 Speed-up Techniques in Apriori

Several speed-up techniques are implemented and evaluated. The first one of them is the removal of infrequent elements from a transaction. Bodon [5] has called this a *filtered transaction*. Since each transaction and its elements are visited many times in the support count, removing infrequent elements from transactions has a big impact on the performance of the Apriori algorithm. It should also be noted that the infrequent items are not required in the rest of the processing. We can remove the infrequent elements from transactions as soon as we finish the first iteration of the algorithm, which is finding the frequency of one element candidates. Let t be a transaction composed of {A, B, C, D}. If C and D are found to be infrequent the filtered transaction t is {A, B}.

As the second speed-up technique, for the order of elements in the trie, two options can be discussed: frequency based ordering and others (lexicographic, random etc.)

We can build the trie with both ascending and descending frequency orderings. According to Bodon [5] the ascending frequency tries were faster in Apriori, while memory requirement is the opposite. The hypothesis is that the ascending frequency ordering has the advantage of having rare elements closer to the root and breaking from the loop in earlier steps during support count. Descending frequency ordering carries the advantage of being a smaller trie, with common elements closer to the root, hence it occupies less memory.

The third speed-up is on the calculation of the support count. During the support count of a k item subset in a transaction t, assume that at a certain depth d after j^{th} element in t, we need to check for items at i^{th} position in t such that $j < i < |t| - k + d + 1$. This is because we need at least that amount of items to complete the support count for that subset in the trie. This is called an *early breakup* from the support count function.

As the fourth speed-up, binary search, reverse binary search and simultaneous traversal are tested. While checking for n element subsets for support count, it is possible to search for items to be present in the candidate trie, or work the other way around; look for the presence of n element subtries or paths to be present in the transaction. Although the result is the same, the search space is different and either we search for transaction elements in the trie, or trie elements in the transaction. The latter is called a *reverse binary search*. Simultaneous traversal keeps two iterators, one in the trie and the other in the transaction. The iterator that points to the smaller item is incremented, until a match is found.

The fifth speed-up technique is as follows: During support count using binary search and lexicographic ordering, support count in the trie for a subset of a transaction, after searching for an item, for the next one, we need to look for items that are lexicographically later than the previous item searched. We call this a *lower bound* for the next element. This technique makes the search space smaller.

Using a trie, the candidate generation step is simple. After reaching the deepest level generated in the previous iteration, we just need to combine the elements while taking into account their orderings. One other advantage of using trie is that it is possible to combine the elimination of branches with low support count and candidate generation in one function. In addition to this, in order to save memory, a single trie can hold both frequent itemsets and candidates.

4 Evaluation

In this paper an evolutionary development process is taken. The first parallelization trial is sending the candidates separately to slave processes and let them build their own candidate tries, while trying the different frequency orderings. It is called Version 1 and described in the Section 3.2. The results are shown in Figure 3. The frequency based orderings are applied both on the single processor sequential algorithm and the parallel algorithm. The compact parallel uses a single trie to store candidates and frequent itemsets. The dataset consists of 10.000 transactions, randomly created; each transaction has up to 100 elements.

It is seen that the Version 1 cannot scale its performance as the number slave process increase. On the contrary, Version 2 which sends the candidate trie recursively has a more linear performance increase in parallel with the increase in number of slave processes.

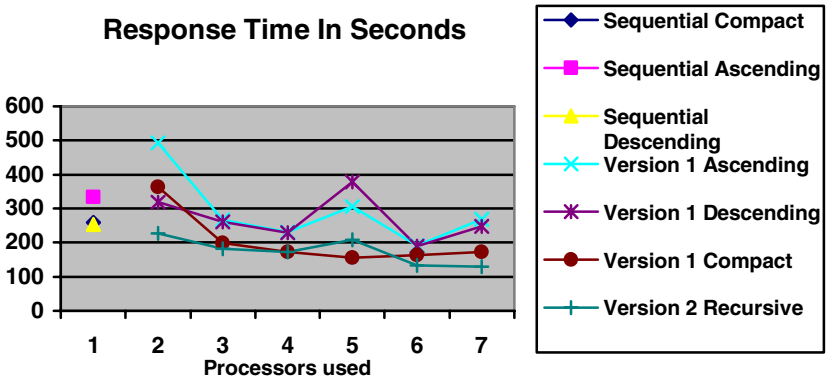


Fig. 3. Comparison of non-recursive (called Version 1) and recursive (called Version 2) trie transfer options. Support threshold 0.45.

When the support threshold is decreased to 0.4, the recursive version has problems of either very long response times, or random PVM terminations. The late responses and terminations are observed usually at certain iterations of the Apriori where candidate numbers generated in the master process is in excess of 25000.

The next version developed, Version 3, aims to decrease the messaging load on the PVM system, by creating a single message composed of a recursive definition of the candidate trie. The performance results of this version are considerably better than the previous one, as shown in Figure 4.

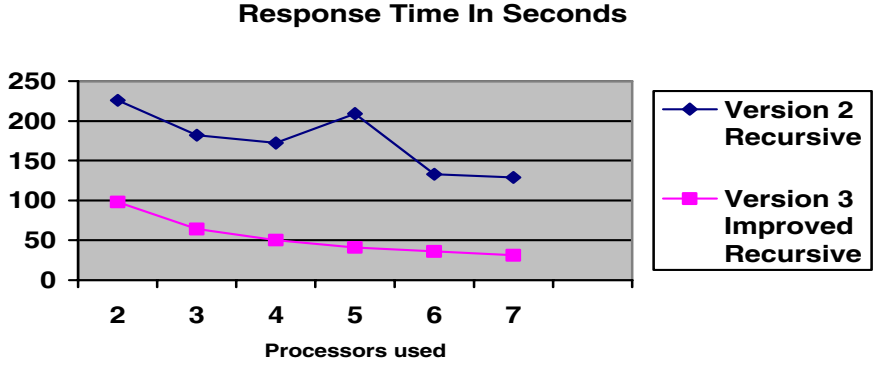


Fig. 4. The improved message passing used in Version 3 shows considerably better response times. Support threshold 0.45. Values for lower support thresholds are not shown in this graphic, since Version 2 cannot produce any comparable times. At support threshold of 0.4 Version 3 has a response time of 59 seconds with 7 slave processes, while PVM terminates with Version 2.

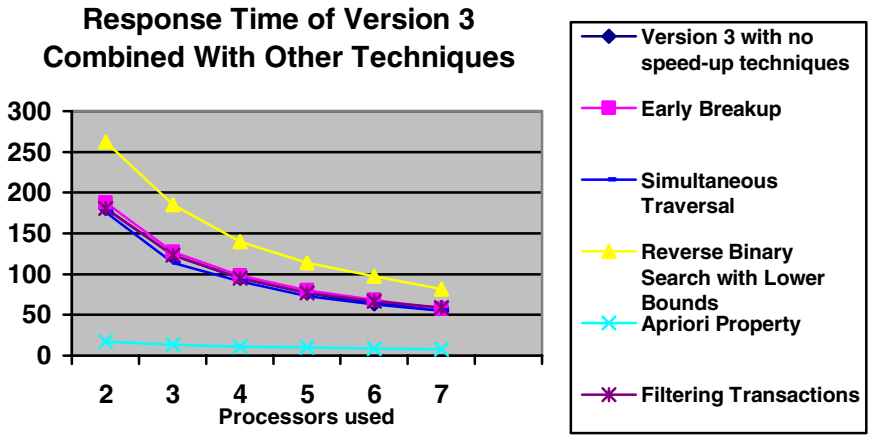


Fig. 5. The same dataset with support threshold now reduced to 0.4 and the additional speed-up techniques applied on Version 3.

Various speed-up techniques are applied to the Version 3 with binary search as the basis and the results are shown in Figure 5. The results to be noted are the effect of the Apriori check on the candidates. This considerably reduces the search space before the support count. In our case, it also reduces the communication time, as the candidate trie is transferred to the slave processes. Simultaneous traversal slightly but consistently outperforms other support count techniques. Reverse binary with lower-bounds is the slowest. Early break up somewhat unexpectedly does not change execution time. It may be due to that fact that this check does also take CPU time, probably overtaking its benefits. Filtering transactions did not change the response time, but it should not be misinterpreted, because effect of this technique is extremely dependent on the data distribution and the support threshold chosen.

5 Conclusions and Future Work

In this paper the trie data structure has been used to test Apriori algorithms's scalability on a parallel computer cluster, while determining contribution of several other techniques. From the experiments, it can be seen that, the communication between the processes in a cluster may be a bottleneck and may be as important as the actual support count function. The improved message passing may be further developed through combining several similar messages into more compact messaging schemas.

Since not all algorithms perform well on all datasets and all threshold values, several processes running different algorithms and switching to the best after few iterations may be evaluated. This is also valid for variations of these algorithms. It is also worthy to implement parallelization efforts on shared memory multi core CPU's in clusters which makes up a mixture of separate memory and shared memory space.

References

[1] Han J. and M. Kamber: Data Mining: Concepts and Techniques Morgan Kaufmann Publishers, August 2000. ISBN 1-55860-489-8
[2] Agrawal R, Srikant R. "Fast Algorithms for Mining Association Rules", VLDB. Sep 12-15 1994, Chile, 487-99, pdf, ISBN 1-55860-153-8.
[3] Jiawei Han, Jian Pei, Yiwen Yin, Mining Frequent Patterns without Candidate Generation, in Proc. of the 2000 ACM SIGMOD Int. Conf. Management of Data, Dallas, TX (2000), pp 1–12.
[4] Mohammed J. Zaki, Scalable Algorithms for Association Mining, In IEEE Transactions on Knowledge and Data Engineering, 12(3) (2000), pp 372–390.
[5] Ferenc Bodon, Surprising results of trie-based FIM algorithms, IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04), in Bart Goethals and Mohammed J. Zaki and Roberto Bayardo editors, CEUR Workshop Proceedings, volume 90, Brighton, UK, 1. November 2004.
[6] F. Bodon. A fast Apriori implementation. In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, 2003.
[7] Ferenc Bodon, A Survey on Frequent Itemset Mining, Technical Report, Budapest University of Technology and Economic, 2006
[8] Han, Karypis, Kumar. Scalable Parallel Data Mining for Association Rules. In Proc. of the ACM SIGMOD Conference on Management of Data, pages 277–288. ACM Press, 1997.
[9] R. Agrawal and J.C. Shafer. Parallel mining of association rules. IEEE Transactions on Knowledge and Data Eng., 8(6):962-969, December 1996.
[10] Ye, A Parallel Apriori Algorithm for Frequent Itemsets Mining. SERA 2006: 87-94

Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging

Aurelien Bouteiller, George Bosilca, and Jack Dongarra

University of Tennessee, Knoxville
bouteill,bosilca,dongarra@cs.utk.edu

Abstract. While high performance computing was eagerly adopted by users as a vehicle for satisfying a growing demand on computational power, some areas are still poorly explored. The MPI paradigm is considered as being the keystone for the large development of the HPC infrastructure over the last decade. However, even today the users have to face the lack of tools able to help increase the stability of the software stack and/or of the applications. In this paper we present and evaluate a tool designed to allow developers to further investigate the execution of parallel applications by enabling them to dynamically move back and forth in the execution timeline of a parallel application. Based on an unobtrusive message logging mechanism, deterministic replay is enforced, leading to a simpler and more efficient way to debug parallel software.

1 Introduction and Motivation

In the last few years, distributed computing platforms have become mainstream High Performance Computing (HPC) platforms. Wide acceptance of parallel computing has been encouraged by the introduction of middle-ware and standards, like the Message Passing Interface (MPI), which help the software designers benefit from the tremendous, but hard to gather, computing power available from parallel machines. Nevertheless, software development has surpassed hardware as the main source of exploitation costs. Finding software flaws is a difficult and time consuming process, even for sequential applications. The fundamental non determinism of distributed systems worsens the picture by adding several failure scenarios involving message races and other time dependent events.

Interactive debugging tools, like `gdb`, have been successfully used for long time to debug sequential applications. It allows for stepping into the program execution, adding breakpoints and reading memory values, which are critical features to find the source of software flaws. Setting a distributed debugging session with `gdb` is time consuming but parallel debuggers like Totalview [1] or DDT [2] solve most of the inconvenience of deploying the debugging framework and give good control over the distributed execution. However, it is very difficult to reproduce the same exact scenario leading to the appearance of the failure, because it requires the programmer to interactively control message interleaves.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 297–306, 2007.

This is a serious limitation, as distributed applications programmers usually need to reproduce on demand the same particular bug scenario several times in order to fully comprehend the mechanisms leading to its appearance.

Several solutions have been proposed to detect programming mistakes in distributed applications. Tools like Umpire [3] propose automatic runtime detection of the most classical MPI bug scenarios. However, a large number of programming mistakes do not belong to checked scenarios and cannot be detected; so remains the need to investigate those bugs. Another popular technique is post-mortem trace analysis [4]. Some traces analyzers even propose deterministic replay of the execution in a simulator [5] with random message payload and delays replacing actual computation. While suitable to tune communication contentions, gathering enough information in order to scope variables at various dates during the execution timeline is impracticable; such a large amount of events would overwhelm typical disk capacities.

All those considerations rise the need for a tool able to enforce deterministic replay of the actual processes of a MPI application. In this paper we present and evaluate a tool based on an unobtrusive message logging mechanism that allows a deterministic replay of the application, at least for applications completely developed over MPI. The same bug scenario can be repeated, and each particular process can be monitored with a debugger (such as `gdb`) until it is fixed. Several message payload strategies allow the user to run only a subset of the processes during a debugging session, and checkpoints are used 1) to migrate the application from an overloaded shared cluster to a more convenient and available set of local machines and 2) to replay a particular time interval of the application's execution without having to restart it from the beginning.

This paper is organized as follows. In the next section, some related works on deterministic replay of distributed applications are discussed. In the third section, the architecture and implementation of our deterministic replay middleware are presented. Some usage scenarios are discussed. The fourth section presents the performance impact of our debugging system. Finally, we conclude and discuss some future works.

2 Related Work

The system community has recently showed some interest in deterministic replay of distributed UNIX applications. In [6] is presented `liblog`, a libc wrapper loaded at runtime that monitors TCP streams, UDP packets, signals, memory allocations and enforces a deterministic thread scheduling. Each network packet is annotated with a Lamport clock and thus can be replayed in correct order. Migratable checkpoints are used to move all processes and logs to a centralized replay server. Because high performance networks are accessed directly by the MPI library without involving the TCP stack, `liblog` wrappers are bypassed. Furthermore, centralized replay is not a practical option when considering the typical memory footprint of MPI processes.

Deterministic replay have also been integrated in MPI libraries. Fault tolerant MPI middle-ware, like MPICH-V [7], are using message logging to enforce correct global state after failure recovery. The unpredictable nature of failures requires to log synchronously the message payload on the sender and receive events on a remote stable storage, which incurs unnecessarily high overhead in the context of application debugging. Several papers focus solely on MPI debugging [8,9]; among them [10] supersedes most of other works. It considers two different kind of events, promiscuous receives and non blocking test operations. Indeed, performance evaluations are restricted to a small set of nodes lacking any scalability evaluations and the selected benchmarks are not representative of typical parallel applications. Performance overhead of all the proposed implementations is high, typically doubling the point-to-point latency when trace collection is activated. Data-replay [11] allows to debug only faulty processes. However message payload logging incurs up to 30% performance degradation compared to event logging and the log amount reaches typical cluster nodes memory size, even for small applications. Such performance degradation might prevent occurrence of races that lead to application failure.

Because of the layer where sits our implementation, trace collection is less obstructive than previous works. Our replay mechanism is decentralized and we provide scalability evaluation up to thousand of nodes. Checkpoints are used to migrate processes to more available resources and to jump directly at the interesting time interval; extra checkpoints can be set anytime during the interactive debugging session to further investigate a particular time step. Receiver based message logging allows to run only suspected processes and their neighbors without full application deployment, and without disturbing event ordering during trace recording.

3 Deterministic Replay

Distributed applications are especially difficult to debug because of message races and time dependent events that raise some uncommon failure scenarios. Considering the level of performance reached by MPI communications, i.e. very low latency and high throughput, it is challenging to design a deterministic replay framework that does not disturb the behavior of the monitored application. This leads to better identify the main sources of non determinism in MPI applications before describing efficient mechanisms to repeat the execution timeline.

3.1 Non Deterministic Events in MPI Applications

Event Model. Each computational or communication step of a process is considered as an event. An execution is an alternate sequence of events and process states; the effect of an event on the preceding state leading the process to the new state. Events can be classified into two categories: deterministic and non deterministic. A deterministic event is an event that always occurs (in every possible execution) from the same state, while a non deterministic event does not

hold this property. As a parallel application is by nature loosely synchronized, there exists no direct time relationship between events occurring on different processes.

In event logging, processes are considered as *Piecewise deterministic*: only sparse non deterministic events occur, separating large parts of deterministic computation. In this paper, we only focus on network, MPI related, events; internal events are usually driven by the determinist code-flow in MPI applications. Even Monte-Carlo algorithms rely on deterministic pseudo-random generators to build a sequence of "random" events. However, if internal non deterministic events should occur anyway, it can be managed using techniques proposed in `liblog`. Considering that any non deterministic event's outcome is stored during the initial execution to a safe repository, a second instance of the program can be forced to replay exactly the same execution.

MPI Communications. For the sake of simplicity, we assume that collective communications are decomposed into point-to-point communications. As imposed by the MPI standard, each communication channel is considered as *FIFO*, but there exist no particular order enforced on messages traveling along different channels. Those are realistic assumptions regarding the most popular MPI implementations: the collective operations are commutative and are deterministic, and the low level device driver takes care of reordering mixed message fragments. However, the major source of non determinism from the MPI application standpoint is the relative order of reception events occurring on a different channel.

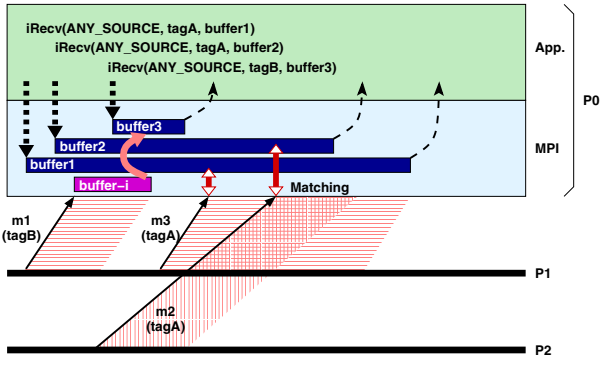


Fig. 1. Basic steps of any source non blocking MPI receptions

Figure 1 shows the basic steps for some non blocking communications. First, the application *requests* a message to be received, specifying message source, tag and reception buffer. When a message arrives from the network, the source and the tag are compared to the pending requests. If the message matches a pending request, like m_3 and m_4, it is directly written in the receive buffer without intermediate copy. Because requests can be ANY_SOURCE, the result of the

matching depends on the order of reception of the first fragment of the message (if m_2 were faster it would have been delivered in buffer$_1$). It is interesting to notice that FIFO property makes ANY_TAG messages deterministic. Next step for the application is to probe for message delivery readiness. The result of such probe functions may depend on the date of message transfer termination, but is not related to the matching order (m_3 is matched first but lasts longer than m_2). This leads to considering two different kinds of events: those related to any-source messages and those related to non deterministic delivery probes.

3.2 Enforcing Deterministic Replay with Message Logging

Any Source Receptions. Any source receptions replay is managed in the iRecv, Recv, and Start MPI functions. At delivery time, the request contains two fields: first is the source as specified by the Recv or iRecv function, second is the communication peer specified by the incoming message header. If the any-source flag has been used, a new match event containing the unique request id and the peer is created. As channels are FIFO, peer id is enough to enforce matching order. This event is then stored to a local stable storage. During replay, once the events have been retrieved from the stable storage; all any-source flagged messages are modified at the beginning of Recv or iRecv functions to have explicit peer source.

Non Deterministic Deliveries. Non deterministic deliveries are rooted in the iProbe, WaitSome, WaitAny, Test, TestAll, TestSome and TestAny functions. We increase a local clock to assign a unique number to each of those operations. When one or more messages are delivered by this particular operation, we create a new delivery event containing the clock and the list of all unique request identifiers delivered during this particular operation.

During replay, as long as the current probe clock is less than the probe clock stored in the first event to replay, this particular operation has to return that no delivery occurred. When the clocks match, the requests corresponding to the event are completed by calling the blocking deterministic delivery operation with the corresponding parameters.

Message Payload Management. Unlike message ordering, message payloads do not need to be stored because they are regenerated during replay. However, it might be convenient for users to step into one single process without running the whole application on the entire cluster. Receiver-based message logging keeps the message payload of incoming messages alongside the events in the log data; the entire communication can be replayed without involving any other process of the system. However, this mode of operation has a major drawback, the amount of data stored on the receiver might be large, depending on the application behavior. Thus, there is a potential to affect the application execution by increasing the number of cache misses.

Checkpoints. Most production MPI applications have a long execution time. Having to wait for hours to replay the execution before reaching the faulty behavior again is a serious limitation. Checkpoints can be used to restart some processes of the parallel application at a date closer to the occurrence of the bug. Indeed, the only way to avoid payload logging when restarting from checkpoints is to use coordinated checkpoints. This synchronization wave typically alters the communication pattern of the application, preventing the appearance of some types of message race bugs. Users should consider a large checkpoint interval during the first run to decrease the probability of impacting trace generation.

3.3 Implementation in Open MPI and Usage

Figure 2 summarizes the Open MPI communication layers. Dashed components have been added to the core Open MPI architecture to enforce deterministic replay of the process communications. The lowest layer of the Open MPI communication stack, the BTL, is not aware of any MPI semantics; it simply moves a sequence of bytes across the underlying transport. The PML implements all logic for point-to-point MPI semantics including short and long message protocols, control messages, standard, buffered, ready, and synchronous modes. Last, MPI, COLL and TOPO components implement the high level MPI interface, including communicators and collective communications on top of point-to-point operations.

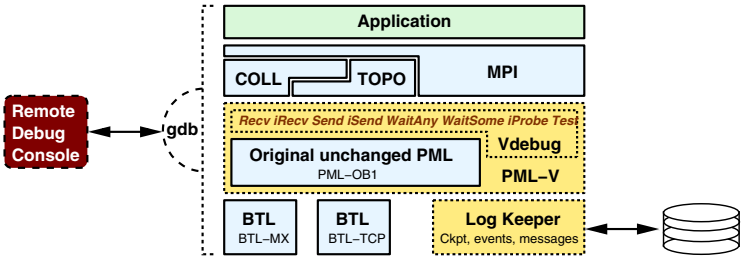


Fig. 2. Added components to the Open MPI architecture to enforce deterministic replay

The PML-V is a parasitic symbiotic component unable to handle itself the usual tasks of a PML. Instead, it loads a monitoring wrapper component: once the PML has been selected, some of the interface functions are replaced with the one provided by a *Vprotocol* component. The `Vdebug` component implements the principles described in the previous section. This modular design does not modify any core Open MPI component and uses the regular optimized BTL devices, preserving a performance profile close to the unmonitored execution.

`Vdebug` includes wrappers for Recv, iRecv, Send, iSend, Start, WaitAny, WaitSome, Test, TestAny, TestSome, TestAll and iProbe functions. Once message logging has been processed, the corresponding PML function is called to achieve

the actual communication. The Log Keeper component logs events to local disk asynchronously in order to avoid running communications at the disk's pace. `Vdebug` uses the Berkeley Lab's Linux Checkpoint/Restart library (BLCR) [12] to take checkpoints and migrate processes.

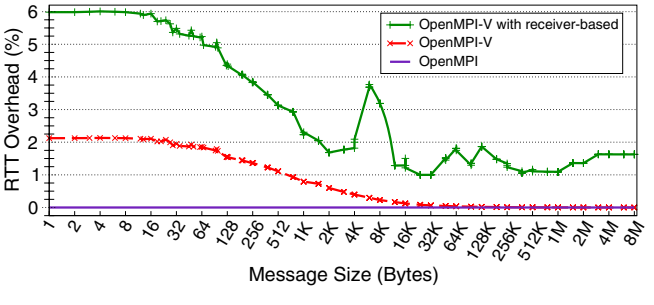
Using Deterministic Replay for Interactive Debugging. Consider a software developer working on application maintenance. The program is compiled and started as usual with the mpicc and mpirun tools. To enable the generation of a set of logs, the users just have to add the "–mca vprotocol_debug_priority 1". The log set is identified by the current jobid, and contains checkpoints and non deterministic events.

If some singular behavior occurs during the run, the user can investigate it by restarting the application and adding the extra "–mca vprotocol_debug_replay jobid" argument. The entire application is restarted on the same set of nodes and a debugging console shows up. From the console, the user can attach a `gdb` to any process, force the generation of receiver-based payload logging for any process, jump to a checkpoint, and adjust the checkpoint interval to a smaller value. Processes are conveniently identified by their rank in the MPI_COMM_WORLD.

When the developer identifies the process and time slot where the flaw appeared, an alternate launch option allows for running a single process from receiver-based logs on the local machine (several instances might run in parallel).

4 Experimental Evaluation

Our experimental testbed is the Lyon Grid5000 cluster. Each node is a dual Opteron 246 (2GHz) with 2GB DDR400 memory. The network interconnect is Myrinet 2000 and the software setup is Linux 2.6.7 with MX-1.03. Benchmarks are compiled using gcc and gfortran 4.1.2 with -O3 and loop unrolling optimizations. We used NetPIPE for round trip time tests and the NAS Parallel Benchmarks to investigate application behavior.



Vdebug Application overhead (%)		
Rcvr-based	None	Yes
BT.B.64	0	6.4
LU.B.64	0	2.2
MG.B.64	2.2	4
CG.B.64	0.8	4
FT.B.64	0	7.3
SP.B.64	0	6.4
EP.B.64	0	0

Fig. 3. Myrinet 2000 performance comparison between core Open MPI and Open MPI-Vdebug with and without receiver-based payload message logging

Figure 3 presents performance on Myrinet for core Open MPI and Open MPI-Vdebug with and without receiver-based message payload logging. On the round

trip time test, the performance penalty of logging the non deterministic events is barely noticeable: the latency is increased by a mere 0.11μs, while, as expected, no impact is visible on the bandwidth. Receiver-based message logging incurs little overhead, latency increases by 0.41μs, bandwidth by about 2% due to memory copies. The most important overhead comes from the size of message payload logs that reach more than 350MB on this simple ping-pong test. If memory gets exhausted, which can happen for highly communicative applications, the performance profile is modified and the communications runs at the disk's pace. Performance on the NAS benchmarks confirms these conclusions: event logging have very little impact with a maximum overhead of 2.2% in MG which is very latency sensitive, and almost no overhead on the others. Receiver-based logging shows up to 7.3% overhead due to memory swapping. So it is beneficial that receiver-based message logging is only activated once the determinism of the application is already enforced.

Another important parameter is the size of logs generated by the event logging mechanism. Among the seven NAS kernels we tested, only two involve non deterministic MPI operations: MG and LU.

Table 1. Log size per process (Kb) depending on the number of processes for some of the NAS Benchmarks (log size for other kernels is zero)

#procs	4	8	16	32	64	128	256	512	1024	Average
LU.B	11.2	11.2	11.9	10.6	13.9	19	14.9	14.2	12.7	13.3
MG.B	11.2	11.2	10.8	10.6	10.6	9.8	9.5	9.3	9.2	10.24

Table 1 presents the log size per process when the number of processes increases in LU and MG class B (class A-C show similar behavior). Per process log size remains close to the average of 13.3kB for LU and decreases with the number of processes in MG. This outlines that the log size does not correlates with the number of processes but with the communication pattern of the application. A scalable application typically tries not to increase the number of communication events per process when the number of nodes increases.

When considering the larger data-set of the class C Benchmarks on 1024 processors, LU generates 287kB per computing node of logs and MG 12kB. This is several orders of magnitude below the available 2GB of memory, meaning that it can be safely flushed to disk without altering application behavior.

5 Conclusion

In this paper, we introduced deterministic replay capabilities in MPI to allow interactive debugging of time related and message race dependent software flaws. During a first run, an unobstructive event logging mechanism stores on disk the outcome of non deterministic MPI communications, while deterministic communications are not logged. This allows the programmer to repeat several time a particular bug scenario under supervision of a debugger. Checkpoints are used to decrease the replay time to reach a singular behavior and allows to migrate

processes to more available hosts. The user can enable receiver-based message logging at any time during the debugging session to specifically investigate a particular process without running the whole parallel application.

Overhead on communication performance is barely noticeable and thus is expected to introduce insignificant differences on application behavior. Event logging memory requirement is moderate, while message payload logging, which produces a large amount of data and might change memory constraints of the application, can be enabled on demand, once the deterministic behavior of the application is already enforced.

In future works, we plan to better interoperate with parallel debuggers like TotalView. We also plan to decrease the log size by using trace compression techniques and incremental checkpoints. Last, we plan to investigate our contention that our checkpoint synchronization approach is unobstructive because it uses the Chandy and Lamport algorithm and does not require flushing the network.

References

1. Gottbrath, C.: Eliminating parallel application memory bugs with totalview. In: SC 2006 Proceedings of the 2006 ACM/IEEE conference on Supercomputing p. 210. ACM Press, New York (2006)
2. Rudgyard, M.: Novel techniques for debugging and optimizing parallel applications. In: SC 2006 Proceedings of the 2006 ACM/IEEE conference on Supercomputing, p. 281. ACM Press, New York (2006)
3. Vetter, J.S., de Supinski, B.R.: Dynamic software testing of mpi applications with umpire. In: SC '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing, p. 51. IEEE Computer Society, Washington, DC, USA (2000)
4. Wolf, F., Mohr, B., Dongarra, J., Moore, S.: Efficient pattern search in large traces through successive refinement. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 47–54. Springer, Berlin (2004)
5. Noeth, M., Mueller, F., Schulz, M., de Supinski, B.: Scalable compression and replay of communication traces in massively parallel environments. In: 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), ACM Press, New York (to appear, 2007)
6. Geels, D., Altekar, G., Shenker, S., Stoica, I.: Replay debugging for distributed applications. In: Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USENIX pp. 289–300 (2006)
7. Bouteiler, A., Herault, T., Krawezik, G., Lemarinier, P., Cappello, F.: MPICH-V project: a multiprotocol automatic fault tolerant MPI, vol. 20, pp. 319–333. SAGE Publications, Thousand Oaks (2006)
8. Clemencon, C., Fritscher, J., Meehan, M.J., Ruhl, R.: An implementation of race detection and deterministic replay with mpi. In: Haridi, S., Ali, K., Magnusson, P. (eds.) EURO-PAR '95: Parallel Processing. LNCS, vol. 966, pp. 155–166. Springer, Heidelberg (1995)
9. Kranzlmuller, D., Schaubschlager, C., Volkert, J.: An integrated record&replay mechanism for nondeterministic message passing programs. In: Proceedings of the 8th EuroPVM/MPI Users' Group Meeting, pp. 192–200. Springer, London, UK (2001)

10. de Kergommeaux, J.C., Ronsse, M., de Bosschere, K.: MPL*: Efficient record/replay of nondeterministic features of message passing libraries. In: Margalef, T., Dongarra, J.J., Luque, E. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1697, pp. 141–148. Springer, Heidelberg (1999)
11. Maryama, M., Tsumara, T., Nakashima, H.: Parallel program debugging based on data replay. In: 17th IASTED International Conference on Parallel and Distributed Computing Systems, pp. 151–156. ACTA Press (November 2005)
12. Duell, J., Hargrove, P., Roman, E.: The design and implementation of berkeley lab's linux checkpoint/restart. Technical Report LBNL-54941, Berkeley Lab (2003)

Extended MPICC to Generate MPI Derived Datatypes from C Datatypes Automatically

Éric Renault

GET / INT — Samovar UMR INT-CNRS 5157
9 rue Charles Fourier
91011 Évry, France
eric.renault@int-edu.eu

Abstract. More and more MPI programs are developed by people who are not experienced parallel programmers; many others are automatically generated by specific frameworks. For both cases, MPI derived datatypes are difficult to handle. This article presents MPIECC, the MPI Extended C Compiler, which aims at introducing a new operator in the MPI API. This new operator is used to automatically translate C datatypes to MPI derived datatypes including pointers and nests of arrays and structures.

Keywords: MPI, Datatypes, Automatic translation.

1 Introduction

As applications are getting larger everyday, and as their computing time is getting longer despite the use of more and more powerful processors, developing parallel applications has become necessary in every scientific fields of research. However, not all the people developing these scientific applications are computer scientists with strong skills in parallel computing. For such a user, message-passing libraries are tools used to perform basic operations like sending and receiving messages, and they are not aware of most functionalities of MPI-like message-passing libraries, like their ability to take into account complex datatypes. Other kinds of users are aware they can define and use derived datatypes but consider this is too difficult to handle. The result is that both kinds of users are limited to basic datatypes and are managing the transfer of complex datatypes manually.

In order to ease the development of parallel applications, tools have been developed to allow the parallel execution of sequential programs. Some of these tools are used at compilation to transform a sequential program into a parallel program automatically [1]; others are used at execution time to allow the parallel execution of a set of instances of a single sequential program, as it is done for task parallelism applications [2].

As a result, for people managing complex datatypes on their own and for applications allowing the execution in parallel of programs provided as sequential codes, there is a need for tools that would automatically manage complex datatypes. Some have been developed for MPI like Automap/Autolink [3] and

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 307–314, 2007.

C++2MPI [4]. However, they are usually limited in their ability to handle complex datatypes and their use is quite orthogonal to either the programming language and/or the message-passing library.

This article presents MPIECC, which stands for MPI Extended C Compiler, an efficient wrapper around MPICC able to generate MPI derived datatypes from C datatypes automatically. As a result, it deals with the C programming language only. However, all the solutions provided could be implemented for a Fortran compiler.

The document is organized as follows. The first section presents the new operator we have introduced inside the MPI API. The next section highlights the benefits involved by using this new operator. Section 4 details the architecture of MPIECC, ie. both compile time and runtime parts are described. The last sections before the conclusion are devoted to the presentation of the limits of MPIECC and a comparison to some related works respectively.

2 MPIECC Principle

In order to let MPIECC know an MPI derived datatype has to be generated, the user is expected to tag any C datatypes used for communications. In the program, this operation is performed using a new dedicated operator (`MPI_Typeof`) where the MPI derived datatype is requested. The `MPI_Typeof` operator can be used in a similar way as the `sizeof` operator, except that the MPI derived datatype is returned instead of the number of bytes for the type provided as a parameter. For example, to get the MPI derived datatype associated to type `char` in the C programming language, one can just use `MPI_Typeof ( char )`. Fig. 1 provides a more complete example.

```
struct list {
    int value ;
    struct list * next ;
} ;

int main ( int argc, char ** argv )
{
    MPI_Status status ;
    int rank, size ;
    struct list * data ;

    MPI_Init ( & argc, & argv ) ;
    MPI_Comm_rank ( MPI_COMM_WORLD, & size ) ;
    assert ( size == 2 ) ;
    MPI_Comm_rank ( MPI_COMM_WORLD, & rank ) ;
    MPI_Send ( data, 1, MPI_Typeof ( struct list ), 1 - rank, 0, MPI_COMM_WORLD ) ;
    MPI_Recv ( data, 1, MPI_Typeof ( struct list ), 1 - rank, 0, MPI_COMM_WORLD, & status ) ;
    MPI_Finalize ( ) ;
    return 0 ;
}
```

Fig. 1. Example of code using `MPI_Typeof`

In order to compile the MPI program and take full advantage of the new functionalities, one shall use `mpiecc` instead of `mpicc`. In fact, MPIECC is implemented

as a wrapper around MPICC to deal with the automatic generation of MPI derived datatypes. However, we expect this new functionality to be included as a standard in a future version of MPI.

3 MPIECC Features

At present, the `MPI_Typeof` operator allows MPIECC to add two new features to the original MPI C Compiler. The first one is the ability to transform C datatypes to MPI datatypes and the second one is the ability to manage links between memory areas.

3.1 Transformation of C Datatypes

C datatypes that MPIECC is able to handle are integral types, arrays, structures, pointers and any compositions at any depths. This task is delegated to MPIPP (an MPI Pre-Processor we developed) [5,6].

Each integral type of the C programming language has a corresponding MPI type. As a result, any call to `MPI_Typeof` for these types or any use of these types inside the definition of a more complex type results in the use of the corresponding MPI datatype.

Arrays are collections of data stored in a contiguous memory area. As a result, the definition of an MPI derived datatype composed of an array is performed using the `MPI_Type_contiguous` function.

A structure is a collection of elements each being located the one after the other one. In order to define the MPI derived datatype associated to a given structure, it is mandatory to determine for each field of the structure, its displacement from the beginning of the structure, the MPI type (derived or not) to associate and the number of items to store (if it is an array). Thus, function `MPI_Type_struct` is used to generate MPI derived datatypes for structures.

3.2 Taking Pointers into Account

MPIECC is able to transfer memory areas linked all together using pointers. Fig. 2 presents an example of complex memory structure to be transfered automatically using MPIECC. In this example, let us assume each node is part of a binary tree with a left and a right subtree. For this particular tree, a node may be the left and/or the right node of more than one node. This is for example the case for node 6 which is at the same time the right node of 4 and the left node of 11. It would lead to wrong behaviours if the result of the transfer includes two copies of nodes 6, 7 and 8. Typically, it would not be possible any longer on the target node to perform a comparison to check if the right pointer of 4 is equal to the left pointer of 11. As a result, the source node (resp. the target node) manages a list of sent (resp. received) memory areas to avoid sending twice the same memory area on the sender and automatically link memory areas together on the receiver.

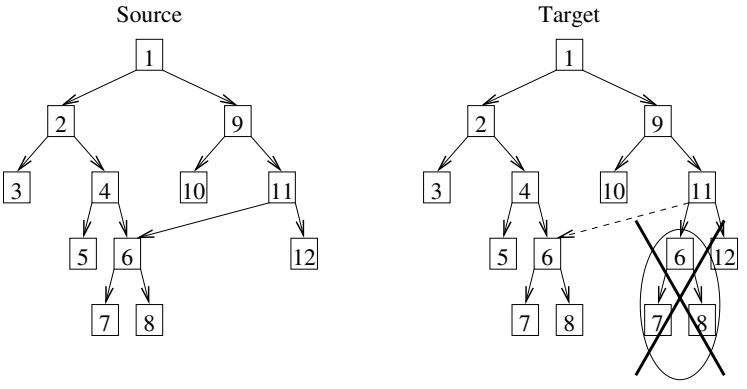


Fig. 2. Example of a complex memory transfer

It is important to note that the reconstruction of the memory structure on the target node is performed in the scope of a single transfer. This means that if the user first sends the content of node 6 (with nodes 7 and 8 transfered implicitly), another transfer including node 6 (for example sending the content of node 4) resends the content of nodes 6, 7 and 8. In fact, as any operations may have been performed between the two transfers, it is not possible to assume that nodes 6, 7 and 8 are still the same as those transfered earlier.

In order to allow MPIECC to determine which memory areas have to be transfered together with the buffer specified by the user, another function (`MPI_Typeof_pointer`) is added to the MPI API. The goal of function `MPI_Typeof_pointer` is to specify the offset from the beginning of the datatype for each pointer. The prototype of function `MPI_Typeof_pointer` is as follows:

```
int MPI_Typeof_pointer ( MPI_Datatype type, int size, MPI_Aint * off, MPI_Datatype * list )
```

where `type` is the MPI derived datatype for which the list of pointers applies to, `size` is the number of pointers in the structure (ie. it is the number of elements of the arrays provided by the last two parameters), `off` is the list of offsets used to indicate where pointers are located from the beginning of the datatype and `list` is the list of MPI derived datatypes, one for each offset.

Function `MPI_Typeof_pointer` is automatically used by MPIECC if necessary to specify the location of pointers inside MPI derived datatypes. As it remains possible for users to define their own MPI derived datatypes, it is also possible to use function `MPI_Typeof_pointer` to specify the location of pointers inside user MPI derived datatypes.

4 MPIECC Architecture

The introduction of these new functionalities inside MPI are performed at two different steps of the program lifetime. The first step occurs at compilation to generate MPI derived datatypes from C datatypes automatically. The second

step occurs at run time where calls to send and receive functions are diverted to take into account pointed memory areas.

4.1 At Compile Time

The definition of MPI derived datatypes is performed at compilation. After the source file has been processed by the C preprocessor, the C file contains the definition of all the datatypes used in the program. As a result, MPIPP (the MPI Pre-Processor we developed to generate MPI derived datatypes from C datatypes) can parse the C file and extract the definition of all user-defined datatypes. Then, MPIPP marks all the datatypes that have been tagged inside the program using the `MPI_Typeof` operator and recursively performs the same operation on all types used for their definition. Finally, the MPI derived datatype for each marked type is generated and calls to the `MPI_Typeof` operator are substituted the corresponding MPI type if the given C datatype is a basic type and a call to the function that defines the MPI derived datatype associated to the given C datatype in the other case. Fig. 3 presents how the output of the C preprocessor is diverted to feed MPIPP and how the output of MPIPP is returned in the usual compilation chain.

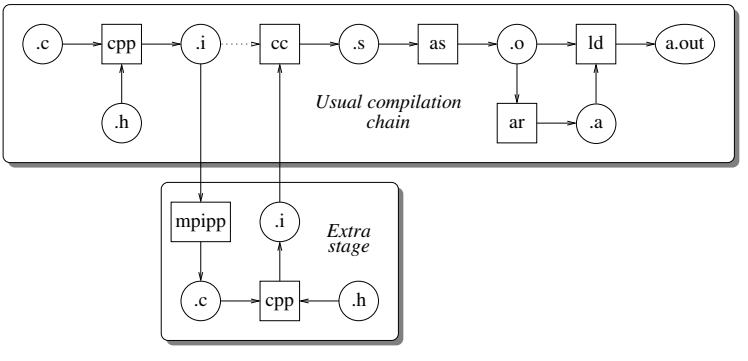


Fig. 3. Introduction of MPIPP in the compilation chain

4.2 At Run Time

There are several solutions to deal with pointers. The one we have chosen consists in using the `LD_PRELOAD` facility provided by the ELF linkage format (available on almost all Unix-like systems) in order to divert calls to send and receive functions. One of the main advantages of using the `LD_PRELOAD` facility is that it makes our solution independent from the underlying MPI implementation.

Send and receive functions provided by MPIECC are able to handle pointers in MPI derived datatypes, ie. memory areas pointed to by non-NULL pointers are sent together with the original buffer. Original send and receive functions of the MPI library are only used to send and receive both predefined MPI datatypes and MPI derived datatypes (with pointers coded as intergers for the data transfer).

5 Limits of MPIECC

MPIECC is able to deal with almost any C datatypes. This section is devoted to highlight the limitations of the present implementation. One can remark that all these limitations are not due to the implementation of MPIECC but they are due to a lack of information at compilation. The only way to bypass these limitations is to get knowledge from the semantic of the program at runtime or ask the programmer to provide information on how these data structures are used.

Unions. Regardless structures, MPI provides no specific function to deal with unions. A solution is provided in [7] but an extra integer is required to select the good MPI derived datatype in an array of MPI derived datatype. Therefore, as this integer must be managed by the user, it is not possible to generate an MPI derived datatype from a C datatype including a union automatically without any extra information.

Void Pointers. MPIECC isable to handle any kind of pointers as long as the datatype of the pointed area is know. The problem with void pointers is that the datatype (and therefore the characteristics like the size) of the pointed area is not defined. It is the programmer responsability to make sure the use of this area is correct. As a result, it is not possible to generate an MPI derived datatype from a C datatype including a void pointer.

Dynamically Allocated Arrays. Pointers in the C programming language are used in order to refer to a memory area. However, even if the datatype of the pointed memory area is known, there is no information to specify whether this memory area stores a single element of that type or several ones. Therefore, without extra information, it is not possible to determine if a memory region

Table 1. Comparison of functionalities

	AutoMap/AutoLink	C++2MPI	MPIECC
□ Language	C	C, C++	C
□ *Datatypes*			
△ basic	✓	✓	✓
△ *array*			
◊ one dim.	✓	✓	✓
◊ multi-dim.			✓
△ structure	✓	✓	✓
△ *depth of nested*			
◊ structures	1	1	any
◊ arrays	1	1	any
△ *pointers*			
◊ management	using functions		transparent
□ *Translation*			
△ number	1	any	any
△ notification	in comments	`# pragma`	`MPI_Typeof`

pointed to inside an automatically generated MPI derived datatype is a single data or not. As a result, we used as a convention that a pointed to memory area is always composed of a single element.

6 Related works

Table 1 draws a comparison in terms of functionalities between AutoMap/Autolink, C++2MPI and MPIECC. MPIECC provides two main advantages regarding the other works. First, it does not require any function calls or comments, but just calls to the `MPI_Typeof` operator. Second, it allows to take into account any kind of C datatype (including nested arrays and structures).

7 Conclusion

This article presented MPIECC, an extension to the MPI C Compiler that enables the automatic generation of MPI derived datatypes from C datatypes automatically. After introducing a new operator to the MPI API, we presented the architecture of MPIECC and how it handles the creation of new MPI derived datatypes. Finally, we presented the limits of our current compiler and a comparison with both the AutoMap/AutoLink couple and C++2MPI and shows that MPIECC provides more functionalities than the other tools and is able to handle more complex cases.

References

1. Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., Zagorodnov, D.: Adaptive computing on the grid using AppLeS. IEEE Transactions on Parallel and Distributed Computing 14(4), 369–382 (2003)
2. Hadjidoukas, P.E.: A Lightweight Framework for Executing Task Parallelism on Top of MPI. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3241, pp. 287–294. Springer, Heidelberg (2004)
3. Goujon, D.S., Michel, M., Peeters, J., Devaney, J.E.: AutoMap and AutoLink: Tools for Communicating Complex and Dynamic Data-Structures Using MPI. In: Panda, D.K., Stunkel, C.B. (eds.) CANPC 1998. LNCS, vol. 1362, pp. 98–109. Springer, Heidelberg (1998)
4. Hillson, R., Iglewski, M.: C++2MPI: A Software Tool for Automatically Generating MPI Datatypes from C++ Classes. In: International Conference on Parallel Computing in Electrical Engineering, Trois-Rivières, QC,, pp. 13–17. IEEE Computer Society, Los Alamitos (2000)
5. Renault, E., Parrot, C.: Automatic generation of mpi derived datatypes from c datatypes with mpipp. In: Proceedings of the ISCA 19th International Conference on Parallel and Distributed Computing Systems, San Francisco, CA, ISCA, The International Society for Computers and Their Applications (2006)

6. Renault, E., Parrot, C.: Mpi pre-processor: Generating mpi derived datatypes from c datatypes automatically. In: Proceedings of the, International Conference on Parallel Processing Workshops, Colombus, OH (2006) pp 248–254 (2006)
7. Message Passing Interface Forum: MPI: A Message Passing Interface Standard (1995)

Timestamp Synchronization for Event Traces of Large-Scale Message-Passing Applications

Daniel Becker[1], Rolf Rabenseifner[2], and Felix Wolf[1]

[1] Forschungszentrum Jülich, John von Neumann Institute for Computing (NIC)
52425 Jülich, Germany
{d.becker,f.wolf}@fz-juelich.de
www.fz-juelich.de
[2] University of Stuttgart, High-Performance Computing-Center (HLRS)
70550 Stuttgart, Germany
rabenseifner@hlrs.de
www.hlrs.de

Abstract. Identifying wait states in event traces of message-passing applications requires measuring temporal displacements between concurrent events. In the absence of synchronized hardware clocks, linear interpolation techniques can already account for differences in offset and drift, assuming that the drift of an individual processor is not time dependant. However, inaccuracies and drifts varying in time can still cause violations of the logical event ordering. The controlled logical clock algorithm accounts for such violations in point-to-point communication by shifting message events in time as much as needed while trying to preserve the length of intervals between local events. In this article, we describe how the controlled logical clock is extended to collective communication to enable a more complete correction of realistic message-passing traces. In addition, we present a parallel version of the algorithm that is intended to scale to thousands of application processes and outline its implementation within the framework of the SCALASCA toolkit.

Keywords: Performance analysis, event tracing, clock synchronization.

1 Introduction

Event tracing is a frequently applied technique for post-mortem performance analysis of message-passing applications because it can be used to analyze temporal relationships between concurrent activities. Obviously, the accuracy of such analyses depends on the comparability of timestamps taken on different processors. Inaccurate timestamps can not only cause a given interval to appear shorter or longer than it actually was, but also change the logical event order, which requires that a message can only be received after it has been sent. This is also referred to as the *clock condition*. To avoid violations of this condition, the error of timestamps should ideally be smaller than one half of the message latency.

Often, however, the clocks accessible from different processors are entirely non-synchronized or only synchronized within disjoint partitions (e.g., SMP-node or

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 315–325, 2007.

multicore-chip). Clock synchronization protocols, such as NTP [4], can align the clocks to a certain degree, but are often not accurate enough for our purposes. Assuming that all local clocks on a parallel machine run at different but constant speeds (i.e., drifts), their time can be described as a linear function of the global time. This approach is used in the tracing library of the SCALASCA toolkit [2], which performs offset measurements between all local clocks and an arbitrarily chosen master clock once at program initialization and once at program finalization. However, as the assumption of constant drift is only an approximation, violations of the clock condition may still occur.

The *controlled logical clock* (CLC) [6] is a method to retroactively correct timestamps violating the clock condition. As the modification of individual timestamps might change the length of local intervals and even introduce new violations, the correction takes the context of the modified event into account by stretching the local time axis in the immediate vicinity of the affected event. The current CLC algorithm, however, is limited by two factors. First, it covers only point-to-point operations and ignores collective ones. Second, it is a serial algorithm designed for a single global trace file. In this article, we describe how the controlled logical clock is extended to collective communication to enable a more complete correction of realistic message-passing traces. In addition, we present a parallel version of the algorithm that is intended to scale to thousands of application processes and outline its implementation design within the framework of SCALASCA [2], a performance-analysis tool that can be used to automatically identify idle times in event traces of large-scale message-passing programs.

The outline of this article is as follows: In Section 2, we start with a short description of SCALASCA's event model and its parallel trace analysis approach, followed by a review of the basic CLC mechanism in Section 3. In Section 4, we describe our extensions required to handle collective operations. After that, we present the new parallel algorithm design in Section 5. Finally in Section 6, we summarize our paper and give an outlook on future work.

2 Event Model and Replay-Based Parallel Analysis

Because we plan to integrate the extended CLC algorithm with the SCALASCA trace-analysis tool, we describe it in terms of the SCALASCA event model, which is similar to the VAMPIR event model [5], for which the algorithm has been originally designed. As far as message passing is concerned, the two models differ only in the way they express collective communication, which the original algorithm ignores anyway.

The information SCALASCA records for an individual event includes at least a timestamp, the location (i.e., the process) causing the event and the event type. Depending on the type, additional information may be supplied. The event model distinguishes between programming-model independent events, such as entering and exiting code regions, and events related to MPI operations. The latter include events representing point-to-point operations, such as sending and receiving messages, and events representing the completion of collective operations.

These collective exit events are specializations of normal exit events carrying additional information (i.e., the communicator) that allows identifying concurrent collective exits belonging to the same collective operation instance. Table 1 illustrates the event sequences recorded for typical MPI operations.

To facilitate trace analysis for large numbers of application processes, the SCALASCA analyzer scans the trace data in parallel. After creating one analysis process per (target) application process, the analyzer loads the entire trace data into the potentially distributed main memory and performs a parallel replay of the applications communication behaviour, thereby examining each communication operation using an operation of similar type. During this procedure, the analyzer measures temporal differences both between remote and between local events, which requires the time stamps to be as accurate as possible. The execution time of the analyzer mainly depends on communication, which resembles the original communication of the target application. For details, please see [2].

Table 1. Exemplary event sequences recorded for typical MPI operations

Function name	Event sequence
`MPI_Send()`	(enter, send, exit)
`MPI_Recv()`	(enter, receive, exit)
`MPI_Allreduce()`	(enter, collective exit) for each participating process

3 Controlled Logical Clock

Non-synchronized processor clocks may cause inaccurate timestamps in event traces. A clock condition violation occurs if the receive event of a message has an earlier timestamp than its matching send event. That is, the *happened-before* relation $e \rightarrow e'$ [6] between two events e and e' with their respective timestamps $C(e)$ and $C(e')$ does not hold. A *clock condition violation* between two events is defined as:

$$\exists\, e, e' : e \rightarrow e' \wedge C(e) \geq C(e'). \tag{1}$$

The CLC algorithm restores the clock condition using happened-before relationships between distributed events derived from point-to-point communication event semantics. More precisely, if the condition is violated for a send-receive event pair, the receive event is moved forward in time. The correction is only applied if the trace contains clock condition violations. To preserve the length of intervals between local events, events immediately following or preceding the corrected event are moved forward as well. This adjustment is called forward and backward amortization, respectively. Note that the accuracy of the adjustment depends on the accuracy of the original timestamps. Therefore, the algorithm benefits from weak pre-synchronization, such as the aforementioned linear interpolation. In this section, we review the CLC algorithm including forward and backward amortization. The interested reader can find a detailed description of the CLC algorithm and a review of further synchronization approaches in [6] and [7].

3.1 CLC with Forward Amortization

The CLC algorithm is an enhancement of Lamport's logical clock [3] and was introduced by Rabenseifner [6]. The algorithm requires timestamps with limited errors, which can be achieved through weak pre-synchronization. To denote timestamps computed by CLC, we use the symbol LC'.

In the following, LC' is modeled with t as the wall clock time and $T(t)$ as the global time to which the process clocks $C_i(t)$ $(i = 0..n-1)$ are synchronized. Next, n is the number of processes, e_i^j is the j^{th} event on process i and so $E = \{e_i^j | i = 0..n-1, j = 0..j_{\max}(i)\}$ is the set of all events in the trace. In addition, the set of matching send and receive pairs is defined with

$$M = \{(e_k^l, e_m^n) | e_k^l = \text{send event}, e_m^n = \text{matching receive event}\}. \quad (2)$$

Note that the send event always marks the beginning of a send operation whereas a receive event marks the end of a receive operation. By contrast, e_i^j is an internal event if it is neither a send nor a receive event. Furthermore, δ_i is the minimal difference between two events on process i and $\mu_{k,i}$ is the minimum message delay of messages from process k to process i. Finally, γ_i^j is a control variable with $\gamma_i^j \in [0,1]$. For each process, LC_i' is now defined as

$$LC_i'(e_i^j) := \begin{cases} \max(LC_k'(e_k^l) + \mu_{k,i}, \\ \quad LC_i'(e_i^{j-1}) + \delta_i, \\ \quad LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ \quad C_i(t(e_i^j))) \qquad \text{if } \exists\limits_{e_k^l} (e_k^l, e_i^j) \in M & (3) \\ \max(LC_i'(e_i^{j-1}) + \delta_i, \\ \quad LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ \quad C_i(t(e_i^j))) \qquad \text{otherwise.} & (4) \end{cases}$$

As can be seen, the algorithm consists of two equations. Equation (3) adjusts the timestamps of receive events while Equation (4) modifies timestamps of internal and send events. Note that for each process, the terms $LC_i'(e_i^{j-1}) + \delta_i$ and $LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1})))$ must be omitted for the first event $(j = 0)$.

Through the term $C_i(t(e_i^j))$ in Equation (3) and Equation (4), the algorithm ensures that a correction is only applied if the trace violates the clock condition. The new timestamps satisfy the clock condition, since the term $LC_k'(e_k^l) + \mu_{k,i}$ in Equation (3) ensures that $LC'(e_i^j)$ is put forward compared to $C_i(t(e_i^j))$ if needed in case of a clock condition violation. To ensure that the clock does not stop after a clock condition violation, the term $LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))$ in Equation (3) and Equation (4) approximates the duration of the

original communication after a clock condition violation. This mechanism is called forward amortization.

Moreover, Rabenseifner has shown that γ_i^j with a constant value can cause LC' to be faster than the fastest clock among all process-local clocks C_i [7]. Cyclic changes of physical clock drifts may cause an avalanche effect that enlarges the value of clock corrections and propagates until the end. To avoid this effect, a control loop is used to find the optimal value of γ_i^j. The controller tries to limit the differences between LC' and T, i.e., the controller estimates the output error indirectly because $T(t(e_i^j))$ is unknown. If $1-\gamma$ is chosen smaller than the maximal drift differences, the controller will enlarge $1-\gamma$ (e.g., to 1%) to ensure that any propagation is bounded by this factor. To calculate γ_i^j for each event, the controller requires a global view of the event data. Mainly, γ_i^j is kept less than 1 minus the maximal drift of the clocks, however, in most cases a fixed $\gamma = 0.99$ or 0.999 is good enough because physical clock drifts are normally less than 10^{-4}. For subsequent events of the same process, the term $LC_i'(e_i^{j-1}) + \delta_i$ in Equation (3) and Equation (4) causes LC' to advance at least a small number of ticks δ_i if the controller has reduced γ_i^j to nearly zero. Rabenseifner describes the control mechanism in more detail in [7].

A jump discontinuity in LC' of Δt is caused by the term $LC_k'(e_k^l) + \mu_{k,i}$ in Equation (3) if $LC'(e_i^j)$ of the violating receive event is put forward compared to $C_i(t(e_i^j))$. The term $LC_i'(e_i^{j-1}) + \gamma_i^j(C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))$ in Equation (3) implements a forward amortization of such a jump. That is, the clock LC_i' for subsequent events of process i runs with the speed of C_i reduced by the factor γ_i^j.

3.2 Backward Amortization

Backward amortization is applied to smooth jump discontinuities caused by the first part of the CLC algorithm. This is done by slowly building up the ascension to a jump Δt using a piecewise process-local linear correction in an amortization interval L_A of appropriate size before the violating receive event [7] (Figure 1). The compensation is realized by setting the timestamps forward. If there are no violating send events in the backward amortization interval of a process i, then the dash-dotted linear interpolation can be used. In Figure 1, the horizontal axis represents LC_i^b, which is equal to LC_i' (i.e., the state after forward amortization) but without the jump Δt at event r. The vertical axis shows offsets to LC_i^b after applying different stages of backward amortization. Naturally, the offset at r corresponds to the jump Δt. Note that the smaller the gradient of a clock in this figure, the better the correction and the smaller the perturbation of preceding events. Therefore, the ratio $\Delta t / L_A$ should be only a few percent. Apparently, adjacent clock condition violations cause a larger perturbation.

In addition, not to violate the clock condition, the correction must not advance the timestamps of send events farther than $LC_m' - \mu_{i,m}$ of the corresponding receive event e_m^n of a process m. These upper limits are shown as circled values above the locations of the send events. If these limits are smaller than the dashed-dotted line (here at events s_1 and s_2), then a reduced piecewise linear interpolation function must be used, see the dotted line in Figure 1. As can be

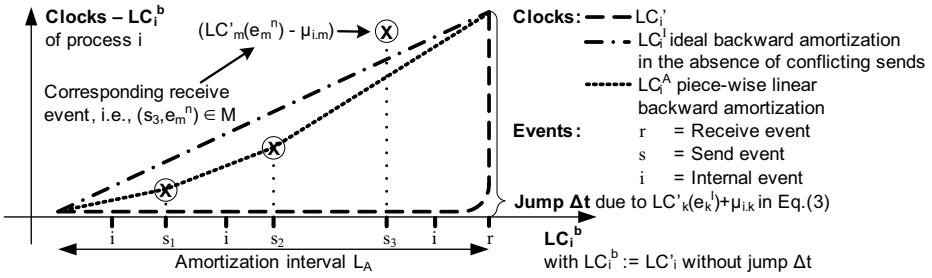


Fig. 1. Algorithm of the backward amortization

seen, the clock error rate is higher than the desired $\Delta t / L_A$ in the interval (s_2, r). For each receive event with a jump, the backward amortization algorithm is applied independently. If there are additional receive events inside the amortization interval during such a calculation step, then these events can be treated like internal events, because advancing the timestamp of a receive event further cannot violate the clock condition.

4 Extended Controlled Logical Clock

Unfortunately, the CLC algorithm in its present state is only designed to correct clock condition violations related to point-to-point communication. Collective communication semantics are ignored. In this section, we extend the algorithm including forward and backward amortization to correctly handle collective communication. Again, we start with considering happened-before relationships among collective communication events. We start with a description of the extended forward amortization followed by the extended backward amortization.

4.1 Extended CLC with Forward Amortization

The CLC algorithm requires the detection of clock condition violations. The happened-before relation is used to synchronize the timestamp of the receive event with the timestamp of the corresponding send event, i.e., the receive event is put forward in time if a clock condition violation has occurred.

A single collective operation can be considered as a composition of many point-to-point communications. Using this model, we determine collective send and receive pairs occurring during a collective operation instance. We distinguish several types of collective operations (e.g., 1-to-N, N-to-1, etc.). Depending on the type, some of the enter events in a collective operation instance can be regarded as send events and some of the collective exit events as receive events.

In the following, we review the different types of collective operations to identify happened-before relationships based on the decomposition of collective operations into send and receive pairs. With S and R we denote the set of send and receive events in a collective operation instance i, respectively. For each call to a collective operation, the set of all send-receive pairs M is enlarged by adding $S \times R$.

1-to-N: One root process sends its data to N other processes. Example are `MPI_Bcast`, `MPI_Scatter`, and `MPI_Scatterv`. S only contains the send event of the root process (i.e., its enter event), whereas R contains receive events from all processes of the communicator (i.e., all collective exit events) with a data length greater zero, i.e., the set may be smaller than the size of the communicator in the case of variable length operations (`MPI_...v`).

N-to-1: One root process receives its data from N processes. Examples are `MPI_Reduce`, `MPI_Gather`, and `MPI_Gatherv`. R only contains the receive event on the root process (i.e., its collective exit event). S is the set of send events (i.e., all enter events) on all processes of the communicator with a data length greater zero. Given that the root process is not allowed to exit the operation until the last process enters the operation, the latest enter event is the relevant send event to fulfill the collective clock condition. Hence, if S contains more than one element, the term $LC'_k(e_k^l)+\mu_{k,i}$ in Equation (3) must be replaced by the maximum of $LC'_k(e_k^l)+\mu_{k,i}$ over all $e_k^l \in S$. That is, Equation (3) must be replaced by

$$LC'_i(e_i^j) := \begin{cases} \max(\ (\max_{\forall e_k^l \text{with}(e_k^l,e_i^j)\in M} LC'_k(e_k^l)+\mu_{k,i}), \\ \qquad LC'_i(e_i^{j-1})+\delta_i, \\ \qquad LC'_i(e_i^{j-1})+\gamma_i^j(C_i(t(e_i^j))-C_i(t(e_i^{j-1}))), \\ \qquad C_i(t(e_i^j))) \quad \text{if } \exists\limits_{e_k^l}(e_k^l,e_i^j)\in M & (3)' \\ \ldots \quad \text{otherwise.} & (4) \end{cases}$$

N-to-N': All processes of the communicator are sender and receiver. Examples are `MPI_Allreduce`, `MPI_Allgather`, `MPI_Alltoall`, and `MPI_Barrier` with N'=N, and the variable length operations `MPI_Reduce_scatter`, `MPI_Allgatherv`, and `MPI_Alltoallv`. S and R are defined by all those enter and collective exit events whose processes contribute input data or receive output data. For a call to `MPI_Barrier`, all processes of the communicator contribute to S and R.

Special cases: For `MPI_Scan` and `MPI_Exscan`, the set of messages added to M cannot be expressed as the Cartesian product $S \times R$. Below, e_k^l refers to the enter event of a collective operation instance and e_i^j refers to the collective exit event and, thus, the set of messages added to M has the form

$$\{(e_k^l, e_i^j) \mid k = 0..N{-}1,\ i = 0..k{-}x\}$$

with $x = 0$ for `MPI_Scan` and $x = 1$ for `MPI_Exscan`.

Independently of collective operation type, it is important to optimize the handling of $S \times R$ in Equation (3). A parallelized algorithm of the extended CLC should attempt to reduce the effort to $\mathcal{O}(\log N)$.

4.2 Extended Backward Amortization

To extend the backward amortization algorithm for collective routines, the upper bounds for the send events (see Figure 1) must be adapted to collective events: If e_i^{j-m} is the send event of a collective routine, an upper bound for the piecewise linear interpolation at e_i^{j-m} is defined by $\min_{e_k^l \in R} LC'_k(e_k^l) - \mu_{i,k}$ with R being the receive event data set defined in Section 4.1.

5 Parallel Timestamp Synchronization

Event tracing of applications running on thousands of processes [8] requires a scalable synchronization scheme. In this section, we present a parallel version of the extended CLC algorithm.

5.1 Pre-synchronization

The accuracy of the CLC algorithm depends on the accuracy of the original timestamps and therefore a pre-synchronization is required. This can be achieved through a linear interpolation where all process-local clocks are mapped onto a single master clock. Given that different clocks vary in offset and drift, offset values between worker processes and one master process measured at program start and at program end are used to find a linear correction function. The offset values are measured using the remote clock reading technique introduced by Cristian [1]. As a byproduct, the minimum transfer delay can be estimated during the offset measurements.

5.2 Parallel Post-mortem Timestamp Synchronization

SCALASCA's replay-based approach of analyzing separate process-local trace files in parallel can handle traces from thousands of processes. We can achieve comparable scalability for the CLC algorithm if we also implement it using a parallel replay. This has the additional advantage that it can be seamlessly integrated into the existing analysis framework.

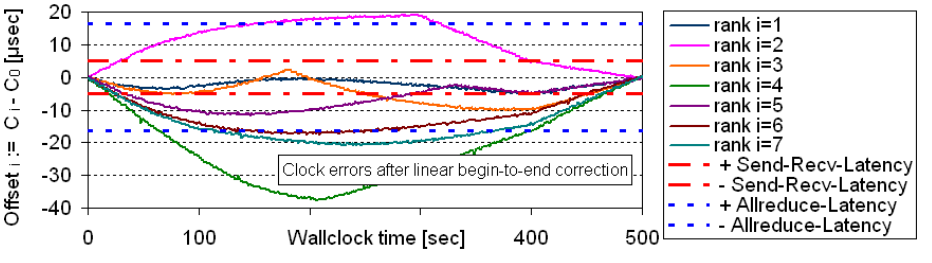


Fig. 2. Non-linear drifts of physical clocks measured on an Infiniband cluster in comparison to Send-Recv and Allreduce latency

Preparation: While each SCALASCA analysis process reads the local trace file of the corresponding application process into memory, the linear correction is applied to all timestamps based on the previous offset measurements at program start and end. The resulting timestamps are taken as the C_i. Inaccurate C_i can occur for two reasons: (i) inaccurate offset measurements and (ii) time-dependant clock drift. Figure 2 shows the non-linear behavior of the clocks C_i after such linear correction on an INFINIBAND cluster. Clock errors are still significantly larger than point-to-point and collective latencies, i.e., violations of the clock condition can still occur.

Logical clock synchronization algorithm: To apply the extended CLC algorithm, a parallel traversal of the event stream is performed. Whenever reaching communication events, the corresponding communication operation is replayed to exchange the timestamps of communication events for their later comparison. For each event, a new timestamp is calculated using the extended CLC algorithm. The comparison between the remote timestamp and the local timestamp is used to find clock condition violations. Depending on the type of the original communication operation, different timestamps are exchanged using different MPI function calls, as listed in Table 2.

Table 2. Timestamps exchanged depending on the type of operation during forward amortization

Type of operation	timestamp exchanged	MPI function
P2P	timestamp of send event	`MPI_Send`
1-to-N	timestamp of root enter event	`MPI_Bcast`
N-to-1	max(all enter event timestamps)	`MPI_Reduce`
N-to-N'	max(all enter event timestamps)	`MPI_Allreduce`
`MPI_Scan`	max(some enter event timestamps)	`MPI_Scan`
`MPI_Exscan`	max(some enter event timestamps)	`MPI_Exscan`

Note, that in Equation (3'), the parallel calculation of the maximum over all corresponding send events ($\max_{\forall e_k^l \text{ with } (e_k^l, e_i^j) \in M} LC'_k(e_k^l) + \mu_{k,i}$) in the case of N-to-1, N-to-N', `MPI_Scan`, and `MPI_Exscan` can not be implemented with the MPI function identified in Table 2 if $\mu_{k,i}$ is not the same for all pairs of processes. Therefore in Equation (3'), $\mu_{k,i}$ must be substituted by $min_{\forall k,i}(\mu_{k,i})$. The exchanged timestamps are based on the LC' values calculated up to the specific event.

The control mechanism used for the controlled logical clock requires a global view of the trace data to calculate γ_i as described in Section 3. Establishing a global view of the trace data is not feasible with the replay-based approach since communication would be required for each single event. Therefore, we eventually have to perform multiple passes until the maximum error e is below a predefined threshold ϵ. For the first pass through the trace files, we propose to use $\gamma = const < 1$, for subsequent passes a $\gamma_{j+1} < \gamma_j$ should be used.

Backward amortization algorithm: The backward amortization requires a second replay of the target application's communication behavior. Timestamps are

Table 3. Timestamps exchanged depending on the type of operation during backward amortization

Type of operation	timestamp exchanged	MPI function
P2P	timestamp of receive event	`MPI_Send`
1-to-N	min(all collective exit event timestamps)	`MPI_Reduce`
N-to-1	timestamp of root collective exit event	`MPI_Bcast`
N-to-N	min(all collective exit event timestamps)	`MPI_Allreduce`
`MPI_Scan`	min(some collective exit event timestamps)	`MPI_Scan`
`MPI_Exscan`	min(some collective exit event timestamps)	`MPI_Exscan`

exchanged at synchronization points of the application. However, as explained in Section 3, the former sender now needs data from the former receiver and so the roles between sender and receiver are switched during the backward amortization. Depending on the type of operation, the collective receiver needs the timestamp of the relevant collective send event which are shown in Table 3. For `MPI_Scan` and `MPI_Exscan`, a communicator with reverse rank ordering must be used. The exchanged timestamps are based on the LC' values after completion of the extended CLC algorithm. After receiving the data, each process temporally stores the timestamps to locally apply the backward amortization if LC' exhibits a jump disconutinity. Note that this happens after the forward amortization has already been applied.

Given that most MPI implementations use binomial tree algorithms to perform their collective operations, our replay-based approach reduces the communication complexity automatically to $\mathcal{O}(\log N)$. Moreover, the stepwise parallel replay during the backward amortization phase could be replaced by a single collective operation per communicator for the entire trace - provided that sufficient memory is available.

6 Conclusion

In this paper, we have extended the CLC algorithm to take collective communication semantics into account so that now a more complete correction of realistic message-passing traces can be achieved. Although the extended CLC algorithm only needs information about the respective event semantics (e.g., root sends to all other processes), we would like to point out that the accuracy of our model could be improved if the MPI-internal messaging inside collective operations was exposed using interfaces such as PERUSE. In this case, the decomposition into (additional) send and receive events is naturally given.

Finally, we have presented a design how the previously sequential algorithm can de parallelized and implemented within the framework of the SCALASCA toolkit. Once we have completed the actual implementation, we will perform a detailed quantitative evaluation using real message-passing codes.

References

1. Cristian, F.: Probabilistic clock synchronization. Distributed Computing 3, 146–158 (1998)
2. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Proc. 13th European PVM/MPI Conference, Bonn, Germany, Springer, Heidelberg (2006)
3. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
4. Mills, D.L.: Network Time Protocol (Version 3). The Internet Engineering Task Force - Network Working Group, March 1992. RFC (1305)
5. Nagel, W., Weber, M., Hoppe, H.-C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer 12(1), 69–80 (1996)
6. Rabenseifner, R.: The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters. In: Proc. 5th EUROMICRO Workshop on Parallel and Distributed (PDP'97), London, UK, pp. 477–484 (1997)
7. Rabenseifner, R.: Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen. PhD thesis, Universität Stuttgart (March 2000)
8. Wolf, F., Freitag, F., Mohr, B., Moore, S., Wylie, B.: Large event traces in parallel performance analysis. In: 8th Workshop Parallel Systems and Algorithms (PASA), Lecture Notes in Informatics, Frankfurt/Main, Germany, March 13-16, Gesellschaft für Informatik (2006), http://icl.cs.utk.edu/projectsfiles/kojak/pubs/pasa06.pdf

Verification of Halting Properties for MPI Programs Using Nonblocking Operations*

Stephen F. Siegel[1] and George S. Avrunin[2]

[1] Verified Software Laboratory, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716, USA
siegel@cis.udel.edu,
http://www.cis.udel.edu/~siegel
[2] Laboratory for Advanced Software Engineering Research, Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA
avrunin@cs.umass.edu,
http://www.math.umass.edu/~avrunin

Abstract. We show that many important properties of certain MPI programs can be verified by considering only a class of executions in which all communication takes place *synchronously*. In previous work, we showed that similar results hold for MPI programs that use only blocking communication (and avoid certain other constructs, such as MPI_ANY_SOURCE); in this paper we show that the same conclusions hold for programs that also use the *nonblocking* functions MPI_ISEND, MPI_IRECV, and MPI_WAIT. These facts can be used to dramatically reduce the number of states explored when using model checking techniques to verify properties such as freedom from deadlock in such programs.

1 Introduction

MPI includes nonblocking communication constructs, such as MPI_ISEND and MPI_IRECV, that can increase the parallelism of a program by allowing processes to overlap computation and communication. But this same parallelism can make programs hard to understand, test, and debug. In principle, model checking techniques [1] can explore all the possible executions of a parallel program and detect problems such as deadlock even if those problems arise only on rare and unusual execution paths, and these techniques have attracted growing interest from the scientific computing community. One of the main difficulties in the application of model checking, however, is the *state explosion problem*—the number of system states that the model checker must examine typically grows exponentially in the number of processes in the program. In particular, for MPI programs the buffering of messages makes an enormous contribution to this state explosion. In [13], we showed that many properties of a class of MPI programs using only the (standard mode) blocking communication constructs (MPI_SEND, MPI_RECV, etc.)

* This material is based upon work supported by the National Science Foundation under Grant No. 0541035.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 326–334, 2007.

could be verified by checking only synchronous executions, thereby eliminating the buffering and greatly expanding the range of model checking. In this paper, we extend those results to the nonblocking communication constructs.

In the next section, we explain how we model MPI programs and state a key lemma. In Sec. 3, we use the lemma to show that checking for deadlock can be restricted to synchronous executions, and in fact to an even more restricted type of synchronous execution. We present a few experimental results showing its impact on model checking using the model checker Mpi-Spin [10] in Sec. 4. The last section presents some conclusions and discusses future work.

2 Traces

In what follows, we let P be a *model of an MPI program* in which the only MPI functions used are MPI_INIT, MPI_FINALIZE, MPI_COMM_RANK, MPI_COMM_SIZE, MPI_ISEND, MPI_IRECV, and MPI_WAIT, and which does not use MPI_ANY_SOURCE. We call such a model *permissible*. (We will see below that the list of permitted functions can be expanded significantly.) The definition of such a model can be made mathematically precise in a variety of ways (see, for example, [10]). However, in the discussion here we will use a somewhat informal notion of model in order to emphasize the essential concepts underlying the theorem and its proof.

In essence, P may be thought of as an abstract version of an MPI program consisting of a fixed number of single-threaded processes. Each process has a unique integer *process ID* (pid); the rank of the process in MPI_COMM_WORLD could be used as the pid, for example. A process of P may use MPI_ANY_TAG and may even make nondeterministic choices. Such choices are common in models of MPI programs that have been created by abstracting the data and variables in the original program. Each process maintains a local *state*, which may be thought of an assignment of values to all variables in the process, including "invisible" variables such as the program counter.

An execution of P is considered to progress in a series of discrete atomic steps, each step corresponding to the execution of a statement by one process in P and possibly modifying the state of that process. In general, there are many different ways P can execute, because the steps from the different processes can be interleaved in various ways and a process can make nondeterministic choices. An execution can consist of a finite or (countably) infinite number of steps. Finite executions can stop at any point—it is not required that the processes terminate or become permanently blocked—and are sometimes called *execution prefixes*.

We think of each execution of P as leaving behind a *trace* which captures all information regarding that execution. We will treat the trace as a sequence of *events* that occur instantaneously and atomically when the processes execute certain actions. Each event contains complete information describing the change made to the system state. The events fall into the following categories:

1. A *post-send* event ps is generated when an MPI_ISEND statement returns. The information incorporated into this event includes the pid of the sending

process, the pid of the destination process, the tag, and some unique identifier for the request object instantiated by the call, which we will call the *request id* (rid).
2. A *post-receive* event pr is generated upon the return of an MPI_IRECV. The event information includes the pid of the process executing the statement, the pid of the source, the tag (or MPI_ANY_TAG), and the rid.
3. An *enter-wait* event ew is generated just before executing a call to MPI_WAIT. The event information includes the pid of the process executing the call and the rid of the request being waited on.
4. An *exit-wait* event xw is generated when a call to MPI_WAIT returns. The event information includes the pid of the process executing the call and the rid of the request. If the request is for a receive operation, the data received is also included.
5. A *term* event term is generated just before a process terminates.
6. A *local* event local is generated for every statement that does not fall into one of the categories above (and hence does not involve communication). The event includes all information necessary to determine the change in local state resulting from execution of the statement.

We need both the ew and xw events since an MPI_WAIT call can block. To simplify the presentation we have not recorded MPI_INIT and MPI_FINALIZE in the trace. We assume those functions are invoked properly by each process.

We say that traces T and T' are *equivalent* if for each process p, the sequence of events from p are identical in the two traces and the state of p after the last event in T is the same as the state after the last event in T'. In particular, the two traces differ only in how events from different processes are interleaved.

Note that, since we have forbidden MPI_ANY_SOURCE, there is a unique way that send and receive operations can be paired by the MPI infrastructure, and this pairing relation is clearly discernible from the trace. An important point about the pairing relation is that it depends only on the order of the posting events within each process, and not in any way on how events from different processes are interleaved. We will say that a ps or pr event in T is *paired in* T if the pr or ps event with which it must be paired also occurs in T.

The rids allow us to determine which posting events correspond to which ew and xw events. Hence every communication involves at most six related entries: the ps and matching pr, the two corresponding ew events, and the two corresponding xw events. Of course, not all six events need occur. It is possible, for example, for a send to post and then complete without a matching receive ever posting, because the message emanating from the send can be buffered. On the other hand, a receive request can never complete if the related send has not posted. In fact, this last restriction is the only barrier to changing the interleaving of events from different processes. This is made precise by the following:

Lemma 1. *Suppose that T is a trace from P, and let e_i and e_{i+1} be consecutive events in T satisfying the following conditions:*

(i) The two events e_i and e_{i+1} do not come from the same process.

(ii) If e_{i+1} is an xw *for a receive request, then e_i is not the matching* ps.

Then the sequence obtained by transposing e_i and e_{i+1} is a trace of P equivalent to T.

Lemma 1 may be taken as an "axiom" following from the informal semantics described in the MPI Standard [4], or may be proved from a formal model such as the one described in [10].

3 Deadlock

Let T be a finite trace of a model P. A process p is in a *potentially blocked state* at the end of T if p has either terminated or the last event for p is an ew for which the corresponding posting event is not paired in T. The motivation for this terminology is the fact that the Standard permits—but does not require—an MPI implementation to block a (standard mode) send operation until the message can be delivered synchronously, i.e., until the receiving process has posted a matching receive. We say that T ends in a *potentially halted state* if at the end of T every process is potentially blocked. Hence in a potentially halted state it is not possible to progress unless, possibly, send operations are allowed to complete without their matching receives having been posted. To distinguish between the "good" case where all processes have terminated normally and the "bad" one where deadlock may occur, we say T ends in a *potentially deadlocked state* if T ends in a potentially halted state and furthermore at least one process has *not* terminated. We say P is *deadlock-free* if it has no trace ending in a potentially deadlocked state.

A trace T is *synchronous* if every xw event on a send request is preceded by the related pr. Hence a synchronous trace represents an execution in which no message buffering was required. We say P is *synchronously deadlock-free* if it has no synchronous trace ending in a potentially deadlocked state.

A synchronous trace $T = e_1, e_2, \ldots$ is *greedy* if, for all i, if e_i is an event from process p and e_{i+1} is an event from process $q \neq p$, then p is in a potentially blocked state at the end of $e_1, \ldots, e_i$. In other words, control switches from one process to another in T only when the first process becomes potentially blocked.

Theorem 1. *Let P be a permissible model of an MPI program and assume that no greedy trace of P ends in potential deadlock. Suppose that T is a finite trace from P ending in a potentially halted state. Then there exists a greedy trace T' from P which is equivalent to T.*

Proof. We will produce the greedy trace T' by modifying the interleaving of T. The modification proceeds in m stages numbered $1, \ldots, m$, where m is the number of events in T. Let $T_0 = T$ and let T_i denote the modified trace after stage i ($1 \leq i \leq m$). We will show by induction on i that T_i is equivalent to T and its prefix of length i is greedy. The case $i = 0$ is vacuously true and the case $i = m$ is the desired result.

Suppose $1 \leq i \leq m$ and let the event sequence T_{i-1} just before stage i be $e_1, \ldots, e_m$. By the induction hypothesis, we may assume that the prefix $R = e_1, \ldots, e_{i-1}$ is greedy. Stage i proceeds as follows. Let p be the process of e_{i-1}. (If $i = 1$ we may let p be any process.) Assume p satisfies the following criterion: there is an event from p in the suffix $S = e_i, \ldots, e_m$, and if the first such event e is an xw then the posting event associated with e is paired in R. Then, by Lemma 1, the sequence T_i obtained by commuting e to the left until it is in position i is a trace equivalent to T_{i-1}, which the induction hypothesis tells us is equivalent to T. Moreover, the prefix of T_i of length i is greedy. So if p satisfies the criterion, we have completed the inductive step.

Now if p fails to satisfy the criterion, then it must be potentially blocked. For suppose that there are no events from p in S. Since T ends in a potentially halted state, that means p must either have terminated in R or ended with an ew for which one of the two related posts does not occur in R. In this case, we may choose any process q satisfying the criterion and proceed as before, and the resulting prefix of length i is still guaranteed to be greedy, and we have again completed the inductive step.

We are left with the possibility that no process p satisfies the criterion. In that case, every process is potentially blocked at the end of R, yet not every process has terminated, since there remain events in S. This means R is a greedy trace ending in potential deadlock, contradicting the hypothesis that no greedy trace ends in potential deadlock. □

Corollary 1. *Let P be a permissible model of an MPI program. Then P is deadlock-free if, and only if, P is synchronously deadlock-free.*

Proof. If P is deadlock-free then no trace ends in potential deadlock, so certainly no synchronous trace does. So suppose P is synchronously deadlock-free. Since any greedy trace is synchronous, no greedy trace of P can end in potential deadlock. Now if P had a trace T ending in potential deadlock, then by Theorem 1, P would have to have a greedy trace ending in potential deadlock, a contradiction. So P has no trace ending in potential deadlock, i.e., P is deadlock-free. □

While we have expressed Theorem 1 using a very limited subset of MPI, it is clear that a number of other functions can be safely added to that subset. For example, MPI_SEND is functionally equivalent to an MPI_ISEND followed immediately by an MPI_WAIT, and so can be included. The same goes for MPI_RECV. MPI_SENDRECV may be replaced by the post of the send, followed by the post of the receive, followed by the two waits. MPI_SENDRECV_REPLACE may be expressed in a similar way, using a temporary buffer to handle the receive. All of the collective functions can be modeled using these basic point-to-point functions (see [11]) and so can be included as well. The functions MPI_WAITALL, MPI_SEND_INIT, MPI_RECV_INIT, MPI_START, MPI_STARTALL, MPI_REQUEST_GET_STATUS, and MPI_REQUEST_FREE are also permissible.

It is instructive to see how the proof fails if P uses MPI_ANY_SOURCE. Consider the code in Fig. 1(a). In the sole synchronous trace of this program, every

```
if (rank==0) { MPI_Recv(...,MPI_ANY_SOURCE,...); MPI_Recv(...,2,...); }
else if (rank==1) { MPI_Send(...,0,...);  MPI_Send(...,2,...); }
else if (rank==2) { MPI_Recv(...,1,...);  MPI_Send(...,0,...); }
```

(a) Counterexample using MPI_ANY_SOURCE and 3 processes

```
if (rank==0) {
  MPI_Isend(...,1,tag0,...,&req[0]); MPI_Isend(...,1,tag1,...,&req[1]);
  MPI_Waitany(2,req,&i,...);
  if (i==0) { MPI_Recv(...,1,tag2,...); MPI_Wait(&req[1],...); }
    else { MPI_Wait(&req[0],...); }
} else if (rank==1) {
  MPI_Recv(...,0,tag0,...); MPI_Send(...,0,tag2,...);
  MPI_Recv(...,0,tag1,...);
}
```

(b) Counterexample using MPI_WAITANY and 2 processes

```
assert (rank == 0 || rank == 1);
MPI_Isend(..., 1 - rank, tag1, &req, ...);
MPI_Test(&req, &flag, ...); MPI_Barrier(...);
if (flag) { MPI_Recv(..., 1 - rank, tag2, ...); }
  else { MPI_Recv(..., 1 - rank, tag1, ...); MPI_Wait(&req, ...); }
```

(c) Counterexample using MPI_TEST and 2 processes

Fig. 1. Counterexamples to Corollary 1 for models using "impermissible" primitives

process terminates normally. If buffering is allowed, process 1 may send both messages, then process 2 may execute to completion, then the receive in process 0 may get paired with the message from process 2, and then process 0 will deadlock at the second receive. In attempting to apply the algorithm from the proof to convert this sequence into a synchronous one, one gets to a point where there is no process satisfying the criterion, but nevertheless the synchronous prefix is not potentially deadlocked. The reason is that the wildcard receive can be paired with a *different* send in order to escape from the deadlock. The proof of Theorem 1 depends on the fact that the way sends and receives are paired cannot be changed.

The functions MPI_TEST and MPI_WAITANY are impermissible. Counterexamples using these are given in Fig. 1(b,c).

4 Application

Theorem 1 has implications for model checking to verify properties such as deadlock-freedom: It is sufficient to use the model checker to explore all synchronous (or just all greedy) executions. If no potentially deadlocked state is found, then P is deadlock-free.

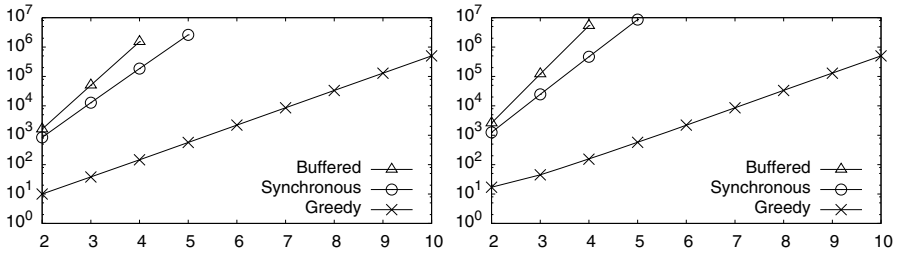
But Theorem 1 applies to more than just deadlock-freedom. Suppose we have established deadlock freedom for P (as above) and we wish to prove some property about the state of P at termination—for example, that the values computed

by P are correct. If we can use the model checker to establish this correctness for all greedy traces, then Theorem 1 implies correctness must hold on any execution. For if there were some execution leading to an incorrect result then Theorem 1 says there must exist an equivalent greedy execution. Since the final values of all variables are the same in equivalent executions, the greedy one will also be incorrect. Hence if the model checker can show that all greedy executions lead to a correct result, we can conclude that all executions do.

Mpi-Spin is an extension to the model checker Spin [2] that adds a number of features corresponding closely to the MPI primitives, making it much easier to model MPI programs for verification. Mpi-Spin can be used to verify properties such as deadlock-freedom and the correctness of the numerical results produced by P. The latter is accomplished by providing Mpi-Spin with both a sequential version of the program, assumed to be correct, and the parallel version and then using Mpi-Spin to show that the parallel and sequential versions must produce the same output on any input. The technique uses symbolic execution to model the numerical operations in the programs and is described in detail in [14]. Mpi-Spin can check properties of programs employing the nonblocking operations but uses a general checking algorithm that does not take advantage of the reductions made possible with Theorem 1. One can force Mpi-Spin to search only synchronous traces, however, by invoking it with the option `-buf=0`. If in addition one encloses the body of each process in an `atomic` block, then only greedy traces will be explored.

To illustrate the impact of the optimizations, we consider Example 2.17, "Use of nonblocking communications in Jacobi computation" from [15]. The program distributes a matrix by columns, maintaining appropriate ghost cell columns. At each iteration, a process first computes the new values for its left- and right-most columns, then posts two send and two receive requests to update the ghost cells, then computes the new values for its interior columns, and finally waits on the four requests before moving to the next iteration.

We considered two properties: deadlock-freedom and the functional equivalence of this parallel program to the simple sequential version (Example 2.12). We scaled n, the number of processes, and set the global matrix size to $3n \times 3n$ and the number of loop iterations to 3. For each n, we used Mpi-Spin to verify



(a) Freedom from deadlock (b) Sequential-parallel equivalence

Fig. 2. The number of states (y-axis) vs. number of processes (x-axis) for verifying two properties of the Jacobi iteration program

each property in three different ways: (1) allowing buffering of messages, (2) allowing only synchronous communication, and (3) allowing only greedy executions. The number of states for each search can be seen in Fig. 2, where each curve shows exponential growth as expected. The synchronous optimization certainly helped but restricting to greedy executions made a dramatic improvement and allowed the verification to scale much further. In fact, our theorem shows that we could look only at the greedy executions in which the processes execute (if not blocked) in some fixed order such as round-robin. This would presumably allow a much greater reduction in the number of states, but we do not see a straightforward way to implement that in Spin.

5 Conclusion

To the best of our knowledge, Matlin, Lusk, and McCune [3] were the first to apply model checking techniques to an MPI problem, using Spin to investigate a component of the MPI implementation MPICH. In [12,13] we showed how Spin could be used to verify properties of simple MPI-based scientific programs; we also presented theorems for countering state-explosion for MPI programs that used only blocking functions and no wildcard receives. These reduction results were generalized to deal with wildcard receives in [9]. Pervez, Gopalakrishnan, Kirby, Thakur, and Gropp [8] used Spin to verify programs that use MPI's "one-sided" operations and found a subtle bug in one such program. The problem of verifying the numerical computations carried out by MPI programs was tackled by Siegel, Mironova, Avrunin, and Clarke in [14], which introduced the symbolic method for establishing the equivalence of sequential and parallel versions of a program. This method was incorporated into the Mpi-Spin tool, which was introduced in [10], along with a technique for modeling nonblocking functions.

Other very recent results appear to be closely connected to this work. In [6], Palmer, Gopalakrishnan, and Kirby introduce a model for a somewhat different subset of MPI and show how dynamic partial order methods allow checking properties by considering only a subset of the possible traces. Pervez, Gopalakrishnan, Kirby, Palmer, Thakur, and Gropp [7] have developed a model checking technique that works directly on source code, bypassing the model construction step. Using the modeling language TLA+, Palmer, Delisi, Gopalakrishnan, and Kirby [5] have developed a formal description of a large portion of the MPI Standard, and have integrated this into model checking tools. In future work, we will explore the connections between these approaches and ours.

In this paper we have generalized some of the earlier reduction theorems to the case of nonblocking functions, and demonstrated how these results can be used to improve the performance of Mpi-Spin. In particular, we have shown that many important properties of MPI programs that use the (standard mode) nonblocking operations can be verified by checking only a special class of executions in which no messages are buffered by the MPI infrastructure. A small example shows that this can provide a very substantial reduction in the resources required for verification, allowing model checking of significantly larger MPI programs. Our

results do not hold for programs that make use of MPI_ANY_SOURCE, MPI_-WAITANY, MPI_WAITSOME, MPI_TEST, MPI_TESTANY, or MPI_TESTSOME. We plan to investigate the generalization of the Urgent algorithm of [9] to handle these constructs.

References

1. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
2. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
3. Matlin, O.S., Lusk, E., McCune, W.: SPINning parallel systems software. In: Bošnački, D., Leue, S. (eds.) Model Checking Software. LNCS, vol. 2318, pp. 213–220. Springer, Heidelberg (2002)
4. Message Passing Interface Forum: MPI: A Message-Passing Interface standard, version 1.1 (1995), http://www.mpi-forum.org/docs/
5. Palmer, R., Delisi, M., Gopalakrishnan, G., Kirby, R.M.: An approach to formalization and analysis of message passing libraries. In: Proceedings of the 12th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS), Springer, Heidelberg (to appear, 2007)
6. Palmer, R., Gopalakrishnan, G., Kirby, R.M.: Semantics driven partial-order reduction of MPI-based parallel programs. In: Parallel and Distributed Systems: Testing and Debugging (PADTAD V), London, (to appear, 2007)
7. Pervez, S., Gopalakrishnan, G., Kirby, R.M., Palmer, R., Thakur, R., Gropp, W.: Practical model checking method for verifying correctness of MPI programs. In: Proceedings of the 14th European PVM/MPI Users' Group Meeting, Springer, Heidelberg (2007)
8. Pervez, S., Gopalakrishnan, G., Kirby, R.M., Thakur, R., Gropp, W.: Formal verification of programs that use MPI one-sided communication. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 30–39. Springer, Heidelberg (2006)
9. Siegel, S.F.: Efficient verification of halting properties for MPI programs with wildcard receives. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 413–429. Springer, Heidelberg (2005)
10. Siegel, S.F.: Model checking nonblocking MPI programs. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 44–58. Springer, Heidelberg (2007)
11. Siegel, S.F., Avrunin, G.S.: Modeling MPI programs for verification. Technical Report UM-CS-2004-75, Department of Computer Science, University of Massachusetts (2004)
12. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In: Graf, S., Mounier, L. (eds.) Model Checking Software. LNCS, vol. 2989, pp. 286–303. Springer, Heidelberg (2004)
13. Siegel, S.F., Avrunin, G.S.: Modeling wildcard-free MPI programs for verification. In: Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05), pp. 95–106. ACM Press, New York (2005)
14. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Using model checking with symbolic execution to verify parallel numerical programs. In: Pollock, L.L., Pezzé, M. (eds.) Proceedings of the ACM SIGSOFT Intl. Symposium on Software Testing and Analysis (ISSTA 2006), pp. 157–168. ACM Press, New York (2006)
15. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI—The Complete Reference, The MPI Core, 2nd edn. MIT Press, Cambridge (1998)

Correctness Debugging of Message Passing Programs Using Model Verification Techniques*

Robert Lovas and Peter Kacsuk

MTA SZTAKI, Laboratory of Parallel and Distributed Systems
H-1518 Budapest, P.O. Box 63, Hungary
{rlovas,kacsuk}@sztaki.hu

Abstract. During the correctness debugging of non-deterministic message-passing programs the software engineers must face the probe effect, the irreproducibility, the completeness problem, and also the large state-space to be discovered. This work attempts to over-come the limitation of existing debugging solutions, and combines the traditional debugging methods with automated modeling and formal verification of parallel programs. The presented debugging framework provides user-friendly facilities for active control and highly automated observation mechanism for message passing programs based on formal methods; Petri-net modeling, partial ordering of state space, and temporal logic assertions.

1 Introduction

One of the main problems in verification and debugging of systems is the so-called state space explosion problem. Partial order reduction [6] is a technique that addresses this problem for concurrent systems by constructing a smaller state space that is searched by the developers or the verification algorithms (e.g. using temporal logic assertions), and considering only a restricted set of behaviors of the system, while guaranteeing that the ignored behaviors do not add any new information.

The presented debugging framework applies a kind of partial order reduction technique, called macrostep-based execution, which was elaborated in the P-GRADE parallel programming environment. P-GRADE [1] provides tools to construct, execute, debug, monitor and visualise message-passing based parallel programs. In P-GRADE development environment, the parallel applications can be constructed based on the syntax and semantics of GRAPNEL hybrid programming language, which provides language elements to express graphically the parallelism, distribution, concurrency, and communication between processes using three hierarchical design levels.

The concept of GRAPNEL language and the application of the debugging method are illustrated through the well-known Producer-Consumer problem. At the top level (see 'Application Window' in Fig. 2), called *application* (or inter-process communication) *level* the outline of the whole application is described graphically with respect to communication connections (channels and ports) among the parallel processes. The

* This work was partially supported by the following EU FP6 grants: CoreGRID (IST-2002-004265), CancerGrid (LSHC-CT-2006-037559), and SEE-GRID-2 (031775).

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 335–343, 2007.

process internal level (see 'Process:…' windows in Fig. 2) is used for describing the inner structure of the individual processes graphically using a control flow-like technique, which describes the message passing related parts of the process. For example, the send (output) and receive (input) operations are represented by labeled rectangles, e.g. 'O1' or 'I1', with the corresponding communication port. At the lowest level of GRAPNEL code, called *textual level*, the developer can define textual C code fragments representing the actual contents of the graphical icons defined at the process internal level.

2 The Concept of Macrostep-Based Execution

The idea of macrostep is based on the concept of collective breakpoints[1], such as the set of CAO^2_1- CAO^2_2- CAO^2_3- CAO^2_4 in Fig. 1, which are placed on the inter-process communication primitives; CAO (output), CAI (input) or CAIALT (alternative input) in each GRAPNEL process. Another example collective breakpoint set can be seen highlighted in Fig. 2, where the program with three processes was stopped at two CAO (labeled 'O1') and one CAIALT operations (labeled 'I1'). These communication actions are indexed by the corresponding process number (lower index), and a serial number (upper index). The set of sequential executed code regions between two consecutive collective breakpoints is called a macrostep. A detailed description of macrostep is given in [2].

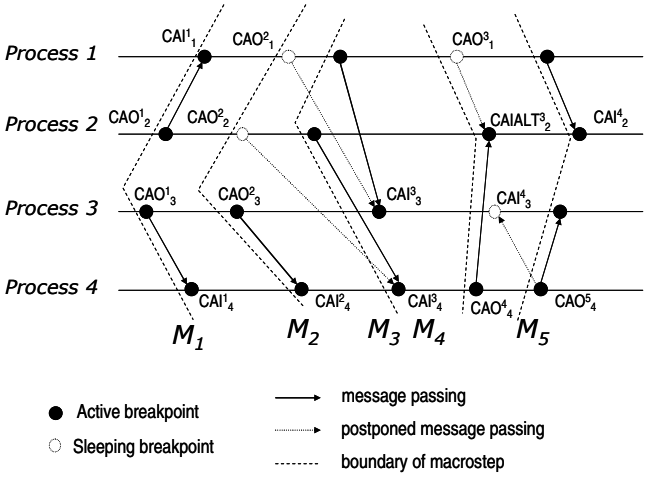


Fig. 1. Illustration for macrostep-based execution

A single breakpoint of the collective breakpoint is called *active* if it was hit in a macrostep and its associated communication operation can be completed (e.g. see

[1] A collective breakpoint consists of a finite number of single breakpoints placed in different processes, and it was hit if (and only if) all the breakpoints belonging the collective breakpoint were hit.

CAO^1_2 in Fig. 1). On the other hand, a breakpoint is called *sleeping* if it was hit in a collective breakpoint but its associated communication instruction cannot be completed only at the next macrostep thus, it will be a part of the next collective breakpoint. For example (see Fig. 1), a send instruction (CAO^2_2) of a given process (*Process 1*) wants to send a message to another process (*Process 4*), but it is communicating with a third process (*Process 3*). That is why, the breakpoint placed at the instruction CAO^2_2 is a sleeping breakpoint and can be found in the next collective breakpoint. Similarly to this, the CAI^3_4 is also a sleeping breakpoint; it must wait for the CAO^4_5.

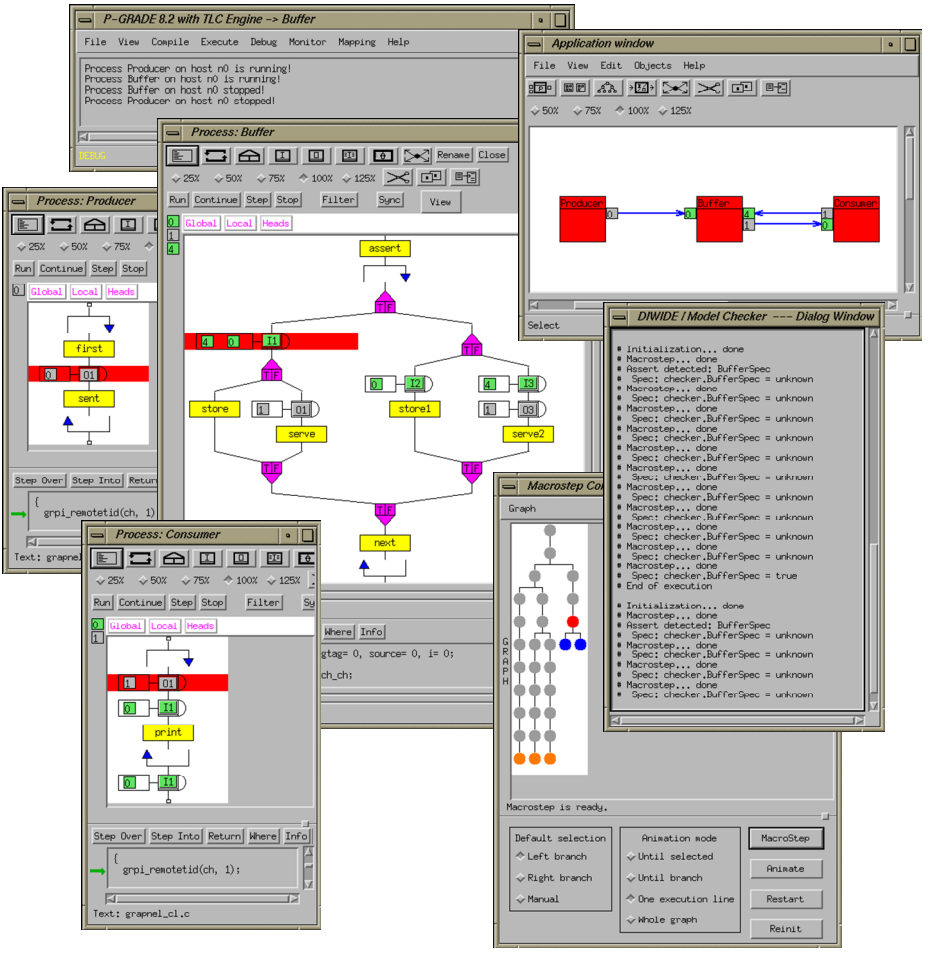


Fig. 2. An example debugging session in the P-GRADE programming environment with the Execution tree of Consumer-Producer application, and with the results of evaluation of temporal logic specification

The execution path is a graph whose nodes represent the macrosteps (i.e. consistent global states) and the directed arcs indicates the possible state transitions between

consecutive macrosteps. The execution tree [2] is a generalization of the execution path (see Fig 2. 'Macrostep …' window as an example); it contains all the possible execution paths of a parallel program assuming that the non-determinism of the current program is inherited only from alternative (CAIALT) receive communications.

3 Coloured Petri-Net Model for Macrostep Based Execution

The formalism of coloured Petri net (CPN, CP net) was chosen for expressing and composing model for GRAPNEL programs in order to simulate the applications, and to create the base of formal description and proof of macrostep-based concept. Two different approaches were taking into consideration to compose CP net models for GRAPNEL programs: the place fusion approach, that models each communication channel explicitly and the environment place approach [3] that models communication channels implicitly, by coupling arc inscriptions. The main disadvantage of environment place approach is that the control of message orderings and the synchronisation dependences between processes cannot be expressed effectively but it is essential in the modelling of the macrostep-based execution.

Following the place fusion approach the application developer can generate automatically the CP net model of GRAPNEL programs based on graph transformation [4]. Briefly, the model of the synchronization channel in the generated CP net works similarly to the traditional CPN model of rendezvous primitive; the corresponding transition is able to fire when both the sender side and the receiver side places hold one token at the matching CAO and CAO operations. The meta-language of Design/CPN tool is used for defining guards for transitions belonging to the conditional and loop constructs in GRAPNEL processes, and compound tokens for description of the local state (i.e. variables) of process. In the CPN model three types of transition can be distinguished and referred later in the proof: transitions belonging to *communications channels of GRAPNEL program* (T_{comm}), the local transitions of individual *GRAPNEL processes* (T_{local}), and special transitions belonging to *alternative input communications* (T_{CAIALT}), which selects the current sender process if there is a race-condition among them. The details can be found in [4].

During the simulation of the CPN model with Design/CPN tool, the Occurrence Graph (OCC graph) of a given model is a directed graph where each $s \in \mathbf{S}$ node represents a possible token distribution (i.e. marking), each $r \in \mathbf{R}$ arc represents a possible state transition between two consecutive token distributions, and

- *enabled(s_i)*: set of (globally) enabled transitions in the given s_i marking. A transition $r \in \mathbf{R}$ is enabled in a state s if there exists a state s' such that $(s; s') \in r$. Otherwise, r is said to be disabled at s. The set of enabled transitions, *enabled(s)* may contain arbitrary types of transitions ($T_{comm,}$ $T_{local,}$ or T_{CAIALT}) depending on the actual local conditions of processes as well as the actual communication interactions.
- *locally_enabled(s_i)*: set of locally enabled transitions in the given s_i marking. A transition $r \in \mathbf{R}$ is locally enabled if the corresponding process enables it (but another process may disable it thus, it may not be part of globally enabled transitions)

- *ample(s_i)* $\subseteq$ enable(s_i): set of *representative* transitions in the given s_i marking, only these transitions are taken into consideration by the generation of state-space during the macrostep execution
- *succ (s_i, j)*: a node of OCC graph where the j^{th} ample transition leads from s_i node.

The macrostep execution reduces the set of enabled transitions, which must be taken into consideration during the construction of global state-space (i.e. the execution tree). This subset of enabled transitions in OCC graph is noted as *ample(s)* in a given marking. The requirements of generation of representative transitions can be summarised as follows: (i) the debugging method must be unaffected by the skipped paths, (ii) the size of graph must be reduced, (iii) the calculation requirements of representative paths must be as low as possible. The ample set generation of macrostep execution follows these rules:

$$\text{enabled}(s_i) = \varnothing \Rightarrow \text{ample}(s_i) = \varnothing \qquad \text{(Rule 1)}$$

$$\text{enabled}(s_i) \cap T_{local} \neq \varnothing \Rightarrow \text{ample}(s_i) = r, \quad \textit{where } r \in (\text{enabled}(s_i) \cap T_{local}) \qquad \text{(Rule 2)}$$

$$\text{locally\_enabled}(s_i) \cap T_{CAIALT} \neq \varnothing \Rightarrow \text{ample}(s_i) = (\text{enabled}(s_i) \cap T_{CAIALT}) \cup \text{enabled}(s_i) \cap T_{partner\_process\ (r)}, \quad \textit{where } r \in \text{locally\_enabled}(s_i) \cap T_{CAIALT} \qquad \text{(Rule 3)}$$

$$\text{enabled}(s_i) \cap T_{comm} \neq \varnothing \Rightarrow \text{ample}(\text{succ}(s_i,j)) = r, \quad \textit{where } r \in \text{enabled}(\text{succ}(s_i, j)) \cap T_{comm} \textit{ and } 0 \leq j < |\text{enabled}(s_i) \cap T_{comm}| \qquad \text{(Rule 4)}$$

In other words; (Rule 1) if there is not enabled transition, the ample set must be empty. (Rule 2) If any of the *local* transitions (T_{local}) is enabled, the ample set must contain only one of these transitions (and nothing else). (Rule 3) If there is no local transition, all enabled *message selection* transitions (T_{CAIALT}) must be included in the current ample set and the enabled (communication) transitions of the processes, which cannot send a message to any alternative input, however it is waiting for it. (4) If only message passing transitions (T_{comm}) are enabled, they must be fired transition by transition.

For practical reasons, the boundaries of macrosteps are defined before applying *Rule 4*, because *Rule 3* is the only one, which can generate an ample set including more than one transition, and the selection of the actual sender processes for CAIALT operations must be allowed to the user. Here, in the execution tree a new branch belongs to each combination of possible sender processes.

4 Correctness of Macrostep-Based Debugging Concept

In this section, the correctness of the macrostep based debugging methodology is presented based on Kripke structures as the common way for the representation of OCC graph and Execution Tree. As the first step, both the Execution Tree (the result of macrostep-based execution) and the Occurrence Graph (corresponding to the entire state space of GRAPNEL program) can be transformed into Kripke structures; KS_e

and KS_p. The labeling of individual nodes can be inherited from the marking of OCC nodes taking into consideration of the compound colored tokens of one selected process P_s, and the actual values of variables of process P_s in case of the Execution tree.

Some key notations must be introduced before the proof: (1) *Invisible transition*: A transition $r = (s, s')$ is called invisible, if $L(s)=L(s')$ i.e. the labeling is not changed between the two transitions. (2) *Stuttering equivalence*: In a path, there can be consecutive states with the same label due to invisible transitions. We can create so-called stuttering blocks from these states, and stuttering equivalent ($\sim_{st}$) paths.

The core mechanism of macrostep-by-macrostep execution can be interpreted as the overlapped execution of independent state transitions. Hence, we can apply some partial ordering techniques and methods on KS_p Kripke structure (derived from the Occurrence Graph of CPN model) in order to get the KS_e, the Execution Tree built by the macrostep debugger.

In order to prove that the Kripke structures are stuttering equivalents to each other, and a particular class of temporal logic expressions (LTL_{-x}) from the program specification can be evaluated on the paths of Execution Tree during the macrostep-based execution; the following conditions (**C0**-**C3**) must be guaranteed according to [6].

4.1 Emptiness

The first condition guarantees that the macrostep algorithm will make progress if the normal search algorithm would:

$$ample(s) = \varnothing \; iff \; enabled(s) = \varnothing \qquad \text{(C0)}$$

The emptiness condition in case of the macrostep algorithm can be proven in the following three steps. (1) The macrostep algorithm generates an empty ample set if there is no enabled transition. It is ensured by Rule 1, which must be applied in all states. Thus, $enabled(s) = \varnothing \Rightarrow ample(s) = \varnothing$ must be true on the entire state space. On the other hand, the transitions can be classified in three types in the CPN model, such as T_{comm}, T_{local}, and T_{CAIALT}. Rules 2-4 guarantee that in each state that $enabled(s) \neq \varnothing \Rightarrow ample(s) \neq \varnothing$, since one of the rules must be always applied if the set of enabled transition is non-empty, and each rule generates a non-empty ample set. Combining the above constraints into one constraint, and after some basic Boolean transformations, we can get **C0** as a result.

4.2 Ample Decomposition

It ensures that any path that is not included in the reduced state-transition graph can be transformed, based on the properties of independent transitions, into a path in the reduced model, and therefore the reduction does not omit any paths which are essential for verification.

In the full state graph, on any path starting from some state *s*,
a transition dependent on a transition from *ample(s)* (C1)
cannot appear before some transition from *ample(s)* is executed.

In practice, instead of expensive algorithms the model checker tools usually take advantage of the specific system structure to generate ample sets of transitions for

which **C1** can be easily guaranteed to hold. Let *ample(s)* be the set of all transitions enabled at s in some set of processes **P** with the following property: no process $P_i \in \boldsymbol{P}$ has a communication transition locally enabled in P_i with a process outside of **P**. The partitioning of the processes in two sets guarantees that by executing transitions outside the ample set it is not possible that a transition dependent on an ample transition will become globally enabled and therefore executed before a transition in the ample set. This is exactly the constraint imposed by **C1**.

The macrostep execution meets **C1** since every rule satisfies it as follows.

- *Rule 1*: There is no enabled transition at all. **P** must be empty.
- *Rule 2*: **P** has always one member process with one enabled local transition, which will form the one-element ample set. This way of ample set generation meets condition **C1**, since this process do not intend to communicate to other processes.
- *Rule 3*: In this case, **P** contains all processes, which have alternative input communication action (CAIALT) as well as all the possible sender process belonging to these alternative input communications. All the enabled CAIALT transitions are the member of ample set. Moreover, the ample set contains the enabled transitions belonging to the sender processes, which are actually not able to send a message to the CAIALT constructions. Thus, condition **C1** is guaranteed.
- *Rule 4*: In this case, we have only pairwised processes, which are ready to exchange their messages and they do not intend to communicate to other processes. Therefore, **P** always contains the two communicating processes, and the ample set contains only one transition, which is a member of T_{comm}, so condition **C1** is guaranteed.

4.3 Invisibility

It must be guaranteed that the specification is not affected, by ensuring that the generated path is stuttering equivalent to the original one, i.e. $q_1, q_2, \ldots, q_n, r \sim_{st} r, q_1, q_2, \ldots, q_n$ [6]. Generally, this aspect is handled by the following condition:

If a state s is not fully expanded,
every transition $r \in$ *ample(s)* must be invisible. (C2)

But a modified version of **C2** can also guarantee this:

If a state s is not fully expanded, every transition,
which is independent from any transition $r \in$ *ample(s)*, must be invisible. (C2')

According to **C2'** the transitions $q_1, q_2, \ldots, q_n$ must be invisible. Hence, $q_1, q_2, \ldots, q_n, r \sim_{st} r, q_1, q_2, \ldots, q_n$ is also guaranteed. Therefore, we can apply either **C2** or **C2'** in order to proof the invisibility condition by each macrostep Rule.

- *Rule 1*: There is no ample set.
- *Rule 2*: The ample set always consists of one local transition $r \in T_{local}$ of a process. If r is not visible (i.e. *not* belonging to the selected process P_s), then **C2** is true. On the other hand, if transition r is visible (i.e. belonging to the selected process P_s), all the other independent transitions $q_1, q_2, \ldots, q_n$ must belong to other (not selected) processes. Thus, transitions $q_1, q_2, \ldots, q_n$ must be invisible, they can not change the labelling according to introduced labelling rules thus, **C2'** is true.

- *Rule 3:* The ample set may consist of transition either $r \in T_{CAIALT}$ or $r \in T_{comm}$. Transition $r \in T_{CAIALT}$ is always invisible; they cannot change the state of control tokens. So if there is not any transition $r \in T_{comm,}$ then **C2** is true. If there is at least one transition $r \in T_{comm}$, we can distinguish two cases. (1) If all transitions $r \in T_{comm}$ are invisible (not belonging to the selected process P_s or communication without CONTROL_PROTOCOL[2]), then **C2** is again true. (2) If one of the transitions $r \in T_{comm}$ belongs to the selected process P_s and communication with CONTROL_PROTOCOL, then this transition will be visible. All the other independent transitions $q_1, q_2, \ldots, q_n$ must belong to other (not selected) processes, which cannot change the labelling, then **C2'** is true.
- *Rule 4*: The ample set always a one-element set containing a transition $r \in T_{comm}$. If one of these transitions $r \in T_{comm}$ belongs to the selected process P_s and communication with CONTROL_PROTOCOL, then this transition will be visible (otherwise **C2** can be applied). The independent transitions $q_1, q_2, \ldots, q_n$ must belong to other (not selected) processes thus, **C2'** is true.

4.4 Cycle Closing Condition

A transition, which is enabled in every state of a cycle in the reduced state space, belongs to the ample set of some state on the cycle. (C3)

The origin of a cycle in the state-space is obviously the loop construct of GRAPNEL programs. The condition **C3** can be fulfilled relying on a simple restriction of GRAPNEL programs, i.e. every loop construct must contain at least one communication action. Consequently, at least one successful communication action, i.e. one macrostep must be done successfully in each cycle involving the application of Rule 4. It ensures that there must be a state, where are only message passing transitions (T_{comm}) enabled in the model. Later, these transitions will form one-element ample sets and will be fired one by one. Therefore, there is no transition, which is always enabled in a cycle, and it is not an element of any ample set.

5 Summary and Related Works

In the last decade, several researchers have developed methods to apply reduction principles in model checking. These techniques include e.g. the stubborn sets method of Valmari [5], or the ample sets method of Peled [6]. These works contain similar ideas, although they differ with respect to the details of the suggested reduction. In our work, the ample sets method was applied because of its similar approach to the original idea of macrostep based debugging implemented in P-GRADE parallel programming environment.

There is a clear analogy between the step-by-step execution mode of sequential programs realized by local breakpoints and the macrostep-by-macrostep execution

[2] If a communication uses this protocol, than the receiver behavior (e.g. number of cycles in loops, or the evaluation of conditions in if-than-else constructs) may be changed according to the received data [4].

mode of parallel programs. This execution mode enables to check the progress of the parallel program at the points that are relevant from the point of view of parallel execution, i.e. at the message passing points. Moreover, based on the presented proof temporal logic assertions can be also evaluated automatically at these points (see 'DIWIDE / Model Checker' window in Fig. 2) as it was demonstrated with the TLC checker [11], and the simulation of future states can steer the debugging session towards to suspicious situations using the CPN model of P-GRADE programs [4].

Concerning the related debugger works, there are some similar advanced debugging frameworks; for example the record & replay tool NOPE (NOndeterministic Program Evaluator) [7] has been developed for testing and debugging of nondeterministic MPI programs within the MAD environment [8], and DDBG [10] together with the STEPS testing tool [9] has been integrated in order to explore systematically the state of PVM programs. However, none of them can support the evaluation of temporal logic specifications against the observed program behavior and the simulation (e.g. based on CPN) of message passing programs are not solved in these systems.

References

1. Kacsuk, P., et al.: GRADE: A Graphical Programming Environment for Multicomputers. Computer and Artificial Intelligence. 17(5), 417–427 (1998)
2. Kacsuk, P.: Systematic Macrostep Debugging of Message Passing Parallel Programs. Future Generation Computer Systems 16(6), 609–624 (2000)
3. Tsiatsoulis, Z., Dozsa, G., Cotronis, Y., Kacsuk, P.: Associating Composition of Petri Net Specifications with Application Designs in Grade. In: Proc. of the Seventh Euromicro Workshop on Parallel and Distributed Processing, Funchal, Portugal, pp. 204–211 (1999)
4. Lovas, R., Vécsei, B.: Integration of formal verification and debugging methods in P-GRADE environment. In: Distributed and Parallel Systems: Cluster and Grid Computing. Kluwer International Series in Engineering and Computer Science, vol. 777, pp. 83–92 (2004)
5. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
6. Peled, D., Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. Software Tools for Technology Transfer 3(1), 279–287 (1999)
7. Kranzlmüller, D., Volkert, J.: NOPE: A Nondeterministic Program Evaluator. In: Zinterhof, P., Vajtersic, M., Uhl, A. (eds.) ACPC 1999 and ParNum 1999. LNCS, vol. 1557, pp. 490–499. Springer, Heidelberg (1999)
8. Kranzlmüller, D., Rimnac, A.: Parallel Program Debugging with MAD - A Practical Approach. In: International Conference on Computational Science 2003, pp. 201–212 (2003)
9. Krawczyk, H., et al.: STEPS - a Tool for Structural Testing of Parallel Software. In the book: Parallel Program Development for Cluster Computing: Methodology. In: Tools and Integrated Environments, ch. 16, pp. 334–354. Nova Science Publishers, New York (2001)
10. Cunha, J.C., et al.: The DDBG Distributed Debugger. In the book: Parallel Program Development for Cluster Computing: Methodology. In: Tools and Integrated Environments, ch. 13, pp. 292–303. Nova Science Publishers, New York (2001)
11. Kovacs, J., et al.: Integrating Temporal Assertions into a Parallel Debugger. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 113–120. Springer, Heidelberg (2002)

Practical Model-Checking Method for Verifying Correctness of MPI Programs

Salman Pervez[1], Ganesh Gopalakrishnan[1], Robert M. Kirby[1], Robert Palmer[1], Rajeev Thakur[2], and William Gropp[2]

[1] School of Computing
University of Utah
Salt Lake City, UT 84112, USA

[2] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA

Abstract. Formal program verification often requires creating a model of the program and running it through a model-checking tool. However, this model-creation step is itself error prone, tedious, and difficult for someone not familiar with formal verification. In this paper, we describe a tool for verifying correctness of MPI programs that does not require the creation of a model and instead works directly on the MPI program. Our tool uses the MPI profiling interface, PMPI, to trap MPI calls and hand over control of the MPI function execution to a scheduler. The scheduler verifies correctness of the program by executing all "relevant" interleavings of the program. The scheduler records an initial trace and replays its interleaving variants by using dynamic partial-order reduction. We describe the design and implementation of the tool and compare it with our previous work based on model checking.

1 Introduction

Parallel programs are notoriously difficult to debug, and MPI programs, particularly those that have intricate control flow or employ relatively new features such as one-sided communication, are no exception. Tools such as MARMOT [7], MPI-CHECK [8], Umpire [16], and Intel Message Checker [2] can detect many errors in MPI programs but do not guarantee that all interleavings of the processes in the program being tested have been systematically examined. While there are an exponential number of such interleavings, partial-order reduction [1, Chapter 10]—a class of methods belonging to the area known as model checking [1]—offers specific approaches to examine only *some* (usually a small fraction) of these interleavings and declare that the effect of examining all the interleavings has been achieved. Partial-order methods are commonly used to verify *models* of parallel programs. For MPI programs, this approach would require that programmers build, either manually or automatically, a *model* (description) of their protocol in a language such as Promela [6], MPI-SPIN [13], or Zing [9]. This model-creation step is known to be tedious and error prone.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 344–353, 2007.

We take the *in situ* approach to model checking, previously demonstrated in the context of many languages, including C programs in tools such as [5,18] and Java programs in tools such as [17]. During in situ model checking, programs written in the target language (usually with some obvious simplifications such as data-range reduction) are directly model checked, without first creating a model. In this paper, we describe our tool that performs in situ model checking of MPI programs that use one-sided communication. We use a *dynamic* version of partial-order reduction (DPOR, [3]) to reduce the number of interleavings, thus being able to, in effect, exhaustively examine all traces of small (but intricate) MPI programs. We call our tool *in situ dynamic partial order*, or ISP. ISP handles many standard MPI communication functions, including `MPI_Barrier`, various flavors of `MPI_Send` and `MPI_Recv`, and some MPI one-sided functions. In this paper we focus on one-sided functions, partly because of the inherent intricacies of handling one-sided communication under in situ scheduling. The complex nature of MPI one-sided communication also forces us to use information specific to the underlying library, MPICH2 in this case. One restriction placed by MPICH2 is that for passive-target one-sided communication, the target process needs to be inside the MPI progress engine in order to process lock requests. We account for this restriction in ISP as described in Section 2. ISP can easily be extended to efficiently handle other MPI implementations.

To motivate the ISP approach, consider the simple MPI program given in Figure 1, executed by two processes P0 and P1. Figure 2 shows how ISP examines two different interleavings of this program. ISP employs the MPI profiling interface, PMPI, to trap MPI calls and hand over control of the MPI function execution to a scheduling process. This *scheduler* can dictate the order in which each process makes MPI calls. We define the block of code starting from the beginning of an MPI call, going forward in the code path including C program statements, and ending at the beginning of the next MPI call to be a *transition* (in our current example, there are no intervening C program statements). We assume that no transition executes infinitely (MPI calls always complete and the intervening C statements have no infinite loops). We also assume that the MPI program is well formed in accordance with the MPI Standard 2.0. The errors detected by ISP are safety properties [1], including deadlocks, violations of `assert` statements placed by the user, and exceptions thrown at runtime.

```
0: MPI_Init
1: MPI_Win_lock
2: MPI_Accumulate
3: MPI_Win_unlock
4: MPI_Barrier
5: MPI_Finalize
```

Fig. 1. Simple MPI program

Given these assumptions, at Step 1, ISP would find processes P0 and P1 to be runnable (`Options`). Assume that ISP randomly chooses `P1`, executing the instruction shown against `P1.1`, which is an `MPI_Win_lock`. At Step 2, P0 and P1 are both runnable again; ISP picks P1, executing `MPI_Accumulate`. Proceeding in this manner, we reach Step 4, where P1 executes `MPI_Barrier`. At this point, the only runnable process would be P0, forcing Steps 5 through 8. The execution of `MPI_Barrier` by P0 results in both processes becoming runnable once again.

Step No.	Proc. Options	First Inter-leaving	Trace due to First Interleaving	Second Inter-leaving	Trace due to Second Interleaving
1:	P0 P1	P1	P1.1: MPI_Win_lock	P1	P1.1: MPI_Win_lock
2:	P0 P1	P1	P1.2: MPI_Accumulate	P1	P1.2: MPI_Accumulate
3:	P0 P1	P1	P1.3: MPI_Win_unlock	P1	P1.3: MPI_Win_unlock
4:	P0 P1	P1	P1.4: MPI_Barrier	P1	P1.4: MPI_Barrier
5:	P0	P0	P0.1: MPI_Win_lock	P0	P0.1: MPI_Win_lock
6:	P0	P0	P0.2: MPI_Accumulate	P0	P0.2: MPI_Accumulate
7:	P0	P0	P0.3: MPI_Win_unlock	P0	P0.3: MPI_Win_unlock
8:	P0	P0	P0.4: MPI_Barrier	P0	P0.4: MPI_Barrier
9:	P0 P1	P1	P1.5: MPI_Finalize	P0	P0.5: MPI_Finalize
10:	P0	P0	P0.5: MPI_Finalize	P1	P1.5: MPI_Finalize

Fig. 2. Interleavings explored for the example in Figure 1

Now ISP picks P1, followed by P0, generating the first interleaving (the last two actions being `MPI_Finalize`).

A naïve implementation of ISP would now backtrack to the *decision point* at Step 9, picking P0 instead of P1, as shown by the second interleaving. We say this is "naïve" because we know that the order in which `MPI_Finalize` is invoked is immaterial. This is precisely what partial-order reduction does: it computes which actions are *commuting actions*, meaning that their interleavings do not produce any semantically observable changes in program outcome. Another commuting pair in the above program would be the two `MPI_Barrier` invocations. Under a naïve approach, an N-way barrier can generate all $N!$ interleavings of the order in which processes encounter the barriers; under partial-order reduction, we can simply generate one interleaving and claim complete coverage.

If our current example is run with MPICH2 with the `MPI_Win_lock` operation specifying an exclusive lock, the only actions that require interleaving are the `MPI_Win_unlock` calls within which shared variable updates take place. For a process accessing the MPI window remotely, only its `MPI_Win_unlock` call modifies the communication window, posting all the accumulated updates within that particular epoch. For this example, the ISP tool would generate two interleavings as opposed to 504 interleavings[1] if we were to use only the in situ feature without DPOR. Since the theory of partial-order reduction is vast, we simply present our assumptions as a table of commuting MPI operations (Figure 6), citing past references [10] based on which such tables can be created. The table can be adjusted to correspond to any MPI implementation of choice, or even to suppress certain interleavings for quicker bug hunting. Also, as opposed to *static* partial-order reduction where the commuting nature of the two `MPI_Finalize` invocations would have been determined while going forward during the first interleaving, we instead follow the *dynamic* approach to partial-order reduction, in which we fully generate the first interleaving and walk up the stack trace and

[1] $2 \times (10!/(5!)^2)$.

mark places where interleavings can be added. Space does not permit a fuller description of DPOR; we note only that it exploits run-time information to effect better reduction (e.g., wildcard communication, as described in [10]).

2 Basics of Scheduling and in Situ Model Checking

In situ model checking lets a *scheduler* control the transitions of the given MPI program. The scheduler opens an array of communication channels (via TCP sockets) through which it receives *appeals* from each process. The pseudocode in Figure 3 captures how the MPI call of a process (generically called `Generic_Func`) is processed. Basically, the MPI function call is intercepted by the profiling library. It then conveys the process id (`pID`), the call type (`Generic_Func`), and the remaining arguments to the scheduler through the `sendToSocket` call. In reply, the scheduler provides either a "go-ahead" or a "loop" to the appealing processes. A loop signal indicates that the appealing process must make an `MPI_Iprobe` call, a side-effect-free mechanism that causes control to enter the MPI progress engine to process all queued-up events within it. `MPI_Iprobe` is needed with MPICH2 in order to cause progress to occur on communication with other processes, because MPICH2 does not use an asynchronous progress thread in its progress engine.[2] When the appealing process receives "go-ahead," it issues `PMPI_Generic_Func`, which then enters the MPI library.

```
MPI_Generic_Func(arg1, arg2...argN) {
sendToSocket(pID, Generic_Func, arg1,...,argN);
while(recvFromSocket(pID) != go-ahead)
MPI_Iprobe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD...);
return PMPI_Generic_Func(arg1, arg2...argN); }
```

Fig. 3. PMPI instrumentation pseudocode

In situ model checking depends on the designer's understanding of how a given MPI library handles each MPI call in terms of the latitude allowed in the MPI standard. For example, MPICH2 treats a `MPI_Win_lock` operation issued from a remote (nontarget) process as a "no operation." However, an `MPI_Win_lock` issued by the target process may cause a lock on the one-sided communication window to be acquired. Let us denote the `MPI_Win_lock` issued from a target as `MPI_Win_lock_T`, from a nontarget as `MPI_Win_lock_NT`, and use the altered names `Win_unlock_T` and `Win_unlock_NT` assuming the same conventions. In our framework, these functions are used to indicate when the "trapped control" comes to the `MPI_Generic_Func` associated with these calls. We also use the notation `PMPI_Win_lock_T` to indicate the `PMPI` call coming after the "trapped" `MPI_Win_lock` call issued by the target, and we similarly use the notations `PMPI_Win_lock_NT`, `PMPI_Win_unlock_T`, and `PMPI_Win_unlock_NT`. These `PMPI` calls signal the point at which the MPI system first knows that these MPI calls are being made. We now present some of the scheduling decisions made by ISP. We explain these with the aid of Figure 1. We rely on the following conventions:

[2] `MPI_Iprobe` does not have a version corresponding to `MPI_Generic_Func`; otherwise, it would cause an infinite loop when these `MPI_Iprobe`s are trapped.

• Because of the assumptions made in Section 1 about *transitions*, we know that each time the scheduler will be handling up to N appeals of the form `sendToSocket(pID, Generic_Func, arg1,...,argN)` (it would be N appeals unless some process has executed a blocking operation).
• The scheduler also keeps track of the lock state of each MPI one-sided window. We will use the terms *window locked* and *window unlocked*.

Consider P0 to be the owner of the window. We call the owner the *target* because that is where all decisions about locking and unlocking the one-sided MPI window are made. Consider the program in Figure 1 run using processes P0 and P1. Specifically, consider the scheduler actions with respect to the following interleaving:
• P0 does `MPI_Win_lock_T`. The scheduler records that the window is locked, and issues a go-ahead, permitting the `PMPI_Win_lock_T` call to be made.
• P1 does `MPI_Win_lock_NT`. The scheduler treats this as a 'no op' (reasons in Section 1) and gives the go-ahead, allowing P1 to make the `PMPI_Win_lock_NT` call.
• P1 does `MPI_Accumulate`. The scheduler gives the go-ahead, allowing P1 to make the `PMPI_Acccumulate` call.
• P1 does `MPI_Win_unlock_NT`. Noting that the window is locked, the scheduler gives the go-ahead, allowing P1 to make the `PMPI_Win_unlock` call. It records that P1 is blocked.
• P0 does its `MPI_Accumulate`, receiving a go-ahead.
• P0 does its `MPI_Win_unlock_T`. Clearly, the scheduler must issue a go-ahead to P0, causing `PMPI_Win_unlock_T` to occur, thus freeing up the window. Note that P1 has already made its `PMPI_Win_unlock_NT` call. However the following race condition could occur: P0 could hurry through the MPI progress engine upon issuing `PMPI_Win_unlock_T`. Suppose P1's lock request reaches P0 only after P0's `PMPI_Win_unlock_T` call has returned. However, since the MPICH2 progress engine has no separate thread to grant locks, P1's successful acquisition of the window is at the mercy of P0 entering the progress engine again, which happens when P0 executes its `PMPI_Barrier` call.
• For simplicity, our scheduler is implemented in so that it moves only one process at a time—in the current example, after we let go P1, we await P1 to make its next MPI command appeal before entertaining any other process.
• However, if we keep P0's appeal in abeyance, the following deadlock might occur: The `PMPI_Win_unlock_NT` can cause an event to be placed in the target's event queue. These events are processed only when the progress engine is entered. Since we have kept P0 in abeyance, however, the progress engine won't be entered.
• Instead of keeping P0 in abeyance, we keep sending "loop" to P0, which causes the `IProbe`'s to be issued. This ensures that P1's event will be processed, causing P1 to reach its next MPI command, at which point we can stop sending "loop" to P0.

```
1  S.add_last(<0...n-1>)  /* randomly choose a proc to run at each depth */
2  backtrack.add_last(<n-1>)
3  done.add_last(<n-1>)
4  if(!fork()) execlp(MPI program) /* run the given MPI program */
5  make all server connections
6  while(backtrack.size() > 0) {
7    current choice = pick randomly from backtrack
8    get readable envelopes for all runnable processes
9    current envelope = envelope for current choice
10   servers[current choice] << goahead /* the chosen MPI process may execute */
11   update block/unblock info for all processes based on current envelope
12   if(chosen process executed MPI_Finalize) {
13     specify that current choice is DONE
14     close(servers[current choice])
15     decrement active procsses }
16   if(active processes != 0) { /* backtrack shows no runnable procs. */
17     if(depth+1 >= backtrack.size()) {
18       rprocs = <all currently runnable processes>
19           if(rprocs.size() == 0)  /* POSSIBLE DEADLOCK */
20             close all socket connections
21             report deadlock and print trace}
22           else { /* current interleaving can be explored further */
23             S.add_last(<all runnable procs>)
24             backtrack.add_last(<S.last.last>) /* randomly choose a proc */
25             done.add_last(<empty>) } }
26     depth++ }
27   else {
28     /* we have gone through one interleaving of the program. Remove all
29        choices from S as well as backtrack until the last decision point.
30        This is where we had more than 1 choice of MPI processes. */
31     while(backtrack.size() > 0 && backtrack.last.size() == 1) {
32       updateBacktrackInfo()
33       S.remove_last()
34       backtrack.remove_last()
35       done.remove_last() }
36     if(backtrack.size() > 0) { /* make sure search is not over */
37       remove most recent choice from backtrack at current depth
38       /* the next interleaving will be forced to take an alternate route */
39       reset checker state for next interleaving
40       if(!fork()) execlp(MPI program)
41       make all server connections
42       depth = 0
43       active procs = n } } }
```

Fig. 4. DPOR-based scheduling algorithm

3 In Situ Model Checking with Dynamic Partial-Order Reduction

The algorithm of Figure 4 shows how ISP exhaustively explores all *relevant* interleavings of the given MPI process as determined by DPOR. The first interleaving is chosen at random by following a standard depth-first search. It is then simply a matter of traversing up the stack, having DPOR identify points where adding interleavings might be useful, and carrying on the search.

The following data structures are used by ISP: **S** contains the set of processes that are able to run at each depth; **backtrack** contains the set of processes that are allowed to run at each depth; **done** contains the set of processes that have been explored at each depth; **servers** is the set of n server connections, one with each MPI process; and **active processes** is initialized to the number of MPI

S		backtrack	First Interleaving	S'		backtrack'	S''		backtrack''
P0	P1	P1	P1.1: MPI_Win_lock	P0	P1	P1	P0	P1	P1
P0	P1	P1	P1.2: MPI_Accumulate	P0	P1	P1	P0	P1	P1
P0	P1	P1	P1.3: MPI_Win_unlock	P0	P1	P0 P1	P0	P1	P0
P0	P1	P1	P1.4: MPI_Barrier	P0	P1	P1			
	P1	P0	P0.1: MPI_Win_lock		P1	P0			
	P1	P0	P0.2: MPI_Accumulate		P1	P0			
	P1	P0	P0.3: MPI_Win_unlock						
	P1	P0	P0.4: MPI_Barrier						
P0	P1	P1	P1.5: MPI_Finalize						
	P1	P0	P0.5: MPI_Finalize						

Fig. 5. DPOR algorithm applied to Example 1

processes n. The most interesting is the *backtrack* set. Readers may view it as a set of sets that keeps track of meaningful interleavings at each depth. The sets S and *backtrack* are almost identical except that S contains the meaningless interleavings as well. Hence, a naïve implementation of ISP would simply discard the *backtrack* set and refer only to S. We now explain how this algorithm works by referring to the interleavings shown in Figure 5. Note that these interleavings correspond to the MPI program of Figure 1.

S is initialized so that its first element contains both P0 and P1. The first element of *backtrack* contains only P1, chosen at random; *depth* is initialized to 0. In lines 12–14 we choose a process randomly from *backtrack* at depth 0. Note that for the entire first interleaving, there will only be one possible choice at each depth. We then indicate to P1 that it may make its MPI call `MPI_Win_unlock`, by answering its appeal with a go-ahead token. ISP must now update its internal bookkeeping information. It does so in line 19 by noting which processes are blocked/unblocked as a result of executing the chosen MPI process. We have now reached a point where *backtrack* will indicate no possible choices in the next step. Line 29 is responsible for calculating the runnable processes so they can be added to the *backtrack* set. Again, both P0 and P1 will be added to S as runnable processes. The last step is to increment *depth* and continue our random depth-first-search algorithm.

The first significant digression from this pattern occurs when P0 calls `MPI_Finalize`. At this point, since both processes have called `MPI_Finalize`, we execute the *else* clause of line 45. The idea is to remove all the choices already made, so that in the next execution of the loop, a different interleaving can be explored. Thus, we remove the `MPI_Finalize` executed by P0. The `updateBacktrackInfo` function then is called on line 51. This function traverses up the set S and identifies any transitions that may need to be interleaved with the `MPI_Finalize` just removed. If any such transitions are identified, the corresponding MPI process is added to the *backtrack* set.

Following our DPOR assumptions, no change results to the *backtrack* set. We continue to remove choices until we reach the `P0.3: MPI_Win_unlock` call. This time, the function `updateBacktrackInfo` updates the *backtrack* set to look like

backtrack' in Figure 5. This indicates that the MPI_Win_unlock functions of P0 and P1 must be interleaved in order to get a different, meaningful interleaving. We continue removing all choices that have already been taken until the backtrack set looks like backtrack". At this point, we are ready to start our search from the beginning.

The DPOR-based algorithm of Figure 4 identifies all such meaningful interleavings and terminates the search either when it encounters a deadlock scenario on line 30 or the search is completed. The commuting MPI operations assumed by ISP are given in Figure 6.

MPIFunctions	*Dependence*
MPI_Init	None
MPI_Send	MPI_Send, MPI_Ssend, MPI_Recv
MPI_Ssend	MPI_Send, MPI_Ssend, MPI_Recv
MPI_Recv	MPI_Send, MPI_Ssend
MPI_Barrier	None
MPI_Win_lock	None
MPI_Win_unlock	MPI_Win_unlock
MPI_Win_free	None
MPI_Finalize	None

Fig. 6. Supported MPI functions

4 Case Study: Byte-Range Locking

Our work in [11] described how we model checked the byte-range-locking protocol presented in [15]. This uncovered a subtle but crucial deadlock bug that had gone unnoticed during testing. With the help of ISP, we successfully caught this bug in the source code of this protocol. We note that no modeling effort and no changes to the source code were required. The results are presented in Figure 7. ISP has been tested on other smaller protocols and has worked as expected. It can be viewed as an exhaustive testing facility that gives the *effect* of examining all interleavings of small but intricate MPI programs.

The most striking feature of these results is that ISP was unable to find this bug without using DPOR. The search algorithm was aborted after it did not finish within 24 hours. Our knowledge of the algorithm allowed us to reduce the search depth and find the bug more quickly. By enabling DPOR within ISP, however, we were able to reproduce the deadlock scenario without having to reduce the search depth. While a hand-written model of the same protocol using SPIN could find the same bug without partial-order reduction [11], with ISP we have eliminated the nontrivial task of modeling MPI programs in Promela. The ISP approach is especially beneficial if the intervening C statements between MPI calls cannot easily be modeled in

Program	*#procs*	*interleavings w/o DPOR*	*interleavings with DPOR*
byterange reduced depth	2	2289	119
byterange full depth	2	-	1522

Fig. 7. Experimental results

Promela, the actual MPI library in use cannot faithfully be modeled, or the error is triggered by a bug in the MPI library.

All our experiments consisted of test programs up to a depth of 20. In other words, they each had fewer than 20 MPI function invocations. Model checking such programs can take anywhere from half an hour to an hour on a single 1 GHz processor with 1 GB of memory.

5 Related Work and Conclusions

Model checking has been used for verifying MPI programs by Siegel et al. in [13,14]. The closest related work to ours is [18], where *distributed* in situ model checking for Pthreads programs has been presented.

In our experience with the byte-range-locking algorithm, the initial program presented in [15] exhibited no discernible bugs, even with conventional testing. However, porting the same program to a laptop caused deadlock. This prompted us to model the protocol in Promela, revealing the bugs reported in [11]. This paper thus comes full circle, and shows that the same bugs can be detected at the C program level *without* model extraction.

Clearly, restarting ISP from `MPI_Init` in order to explore each new interleaving requires a huge overhead, even with partial-order reduction dramatically reducing the number of interleavings. To reduce the overhead, we plan to explore three ideas: (i) divide the program using MPI barriers, and interleave only the code between two subsequent barriers, cutting down the extent of interleavings and also helping to localize errors; (ii) use MPI checkpointing systems (e.g., [4]) to see whether we can checkpoint intermediate states and restart from there as opposed to restarting the search from `MPI_Init`; and (iii) use distributed ISP. We hope to research these topics in the context of ISP.

We would also like to make ISP compatible with all MPI library implementations, not just MPICH2. One issue is that the MPI standard gives too much freedom to implementors. For example, the blocking semantics of `MPI_Send` and some of the one-sided functions are far from being well defined. However, with ISP it is possible to force the underlying MPI library to follow a stricter interpretation of the MPI standard. This approach will allow ISP to be used with any MPI library that conforms to the MPI standard. The code of ISP is available at [12].

Acknowledgments

This work was supported by NSF award CNS-0509379, by the Microsoft HPC Institutes program, and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge, MA (1999)
2. Souza, J.D., Kuhn, B., de Supinski, B.R., Samofalov, V., Zheltov, S., Bratanov, S., Bratanov, S.: Automated, scalable debugging of MPI programs with Intel Message Checker. In: SE-HPCS '05, pp. 78–82 (2005)
3. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121 (2005)
4. Gao, Q., Yu, W., Huang, W., Panda, D.K.: Application-transparent checkpoint/restart for MPI programs over InfiniBand. In: ICPP (August 2006)
5. Godefroid, P.: Model checking for programming languages using Verisoft. In: POPL 97: Principles of Programming Languages, pp. 174–186 (1997)
6. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Reading (2003)
7. Krammer, B., Resch, M.M.: Correctness checking of MPI one-sided communication using MARMOT. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 105–114. Springer, Heidelberg (2006)
8. Luecke, G., Chen, H., Coyle, J., Hoekstra, J., Kraeva, M., Zou, Y.: MPI-CHECK: A tool for checking Fortran 90 MPI programs. Concurrency and Computation: Practice and Experience 15, 93–100 (2003)
9. Palmer, R., Barrus, S., Yang, Y., Gopalakrishnan, G., Kirby, R.M.: Gauss: A framework for verifying scientific computing software. In: Workshop on Software Model Checking. ENTCS 953 (2005)
10. Palmer, R., Gopalakrishnan, G., Kirby, R.M.: Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In: PADTAD (2007)
11. Pervez, S., Gopalakrishnan, G., Kirby, R.M., Thakur, R., Gropp, W.: Formal verification of programs that use MPI one-sided communication. In: EuroPVM/MPI, pp. 30–39 (2006)
12. Preliminary release of the ISP software at `http://www.cs.utah.edu/formal_verification/isp.tar.gz`
13. Siegel, S.F.: Model checking nonblocking MPI programs. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (January 2007)
14. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In: SPIN Workshop, pp. 286–303 (April 2004)
15. Thakur, R., Ross, R., Latham, R.: Implementing byte-range locks using MPI one-sided communication. In: EuroPVM/MPI, pp. 120–129 (September 2005)
16. Vetter, J.S., de Supinski, B.R.: Dynamic software testing of MPI applications with Umpire. In: Proc. of SC2000, pp. 70–79 (2000)
17. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: ASE (September 2000)
18. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Distributed dynamic partial order reduction based verification of threaded software. In: Workshop on Model Checking Software (SPIN 2007) (July 2007)

6^{th} International Special Session on

Current Trends in Numerical Simulation for Parallel Engineering Environments

New Directions and Work-in-Progress

ParSim 2007

Carsten Trinitis[1] and Martin Schulz[2]

[1] Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR)
Institut für Informatik
Technische Universität München, Germany
Carsten.Trinitis@in.tum.de
[2] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
schulzm@llnl.gov

In today's world, the use of parallel programming and architectures is essential for simulating practical problems in engineering and related disciplines. Remarkable progress in CPU architecture (multi- and manycore, SMT, transactional memory, virtualization support, etc.), system scalability, and interconnect technology continues to provide new opportunities, as well as new challenges for both system architects and software developers. These trends are paralleled by progress in parallel algorithms, simulation techniques, and software integration from multiple disciplines.

In its 6^{th} year ParSim continues to build a bridge between computer science and the application disciplines and to help with fostering cooperations between the different fields. In contrast to traditional conferences, emphasis is put on the presentation of up-to-date results with a shorter turn-around time. This offers the unique opportunity to present new aspects in this dynamic field and discuss them with a wide, interdisciplinary audience. The EuroPVM/MPI conference series, as one of the prime events in parallel computation, serves as an ideal surrounding for ParSim. This combination enables the participants to present and discuss their work within the scope of both the session and the host conference.

This year, ten papers with authors in ten countries were submitted to ParSim, and after a quick turn-around, yet thorough review process we decided to accept three of them for publication and presentation during the ParSim session. These three papers show the use of simulation in a range of different application fields including earthquake and turbulence simulation. At the same time, they also address computer science aspects and discuss different parallelization strategies,

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 354–355, 2007.

programming models and environments, as well as scalability. We are confident that this provides an attractive program and that ParSim will yet again be an informal setting for lively discussions and for fostering new collaborations.

Several people contributed to this event. Thanks go to Jack Dongarra, the EuroPVM/MPI general chair, and to Thomas Hérault and Franck Cappello, the PC chairs, for their support to continue the ParSim series at EuroPVM/MPI 2007. We would also like to thank the numerous reviewers, who provided us with their reviews in such a short amount of time (in most cases in just a few days) and thereby helped us to maintain the tight schedule. Last, but certainly not least, we would like to thank all those who took the time to submit papers and hence made this event possible in the first place.

We are confident that this session will fulfill its purpose to provide new insights from both the engineering and the computer science side and encourages interdisciplinary exchange of ideas and cooperations. We hope that this will continue ParSim's tradition at EuroPVM/MPI.

[1] Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. UCRL-PROC-232591.

Gyrokinetic Semi-lagrangian Parallel Simulation Using a Hybrid OpenMP/MPI Programming

G. Latu[1], N. Crouseilles[2], V. Grandgirard[3], and E. Sonnendrücker[2]

[1] INRIA/Scalapplix project & Strasbourg 1 University
LaBRI, 341 cours Libération, 33405 Talence Cedex, France
[2] INRIA/Calvi project & Strasbourg 1 University
IRMA, 7 rue Descartes, 67084 Strasbourg Cedex, France
[3] CEA/DSM/DRFC
Association Euratom-CEA, Cadarache, 13108 St Paul-lez-Durance, France

Abstract. This paper describes a parallel implementation of a numerical solver for the Vlasov equation. The solver is based on a kinetic model describing the motion of charged particles in a plasma. The evolution of the distribution of particles in phase space is computed with an explicit method, and we take into account the self-consistent electric field through the coupling with a Poisson type equation. In this paper, we focus on a recently developed 5D parallel numerical application dedicated to gyrokinetic simulation of tokamak systems and ITG turbulence simulation. A semi-Lagrangian Vlasov solver is used. A specific cubic spline interpolation allows us to formulate a domain decomposition method. A hybrid MPI/OpenMP paradigm was used to benefit from a large number of processors while reducing communication costs.

1 Introduction

Understanding turbulent transport in a magnetically confined plasma is a subject of interest to figure out and optimize physics experiments in the present fusion devices and also to design future reactors. Indeed, the thermal confinement of a magnetized plasma is essentially determined by turbulent heat conduction across the equilibrium magnetic field. In practice, the study of plasma turbulence requires to solve the Maxwell equations coupled to the calculation of the plasma response to the perturbed electromagnetic field. This response can be computed by using either a fluid or a kinetic description of the plasma. The purpose of the code described in this article is to perform kinetic simulations in order to compute accurately the turbulence in nearly collisionless plasmas. In this approach, the time evolution of a particle distribution function is well described by the Vlasov equation. This distribution function $f(\overrightarrow{x}, \overrightarrow{v}, t)$ represents the unknown which depends on the time $t \geq 0$, the position $\overrightarrow{x} \in \mathbb{R}^d$ and the velocity $\overrightarrow{v} \in \mathbb{R}^d$. The quantity $f(\overrightarrow{x}, \overrightarrow{v}, t) d\overrightarrow{x} d\overrightarrow{v}$ represents the number of particles located in a volume $d\overrightarrow{x} d\overrightarrow{v}$ centered around $(\overrightarrow{x}, \overrightarrow{v})$. To describe the most general case, one

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 356–364, 2007.

needs $(\overrightarrow{x}, \overrightarrow{v}) \in \mathbb{R}^d \times \mathbb{R}^d$ with $d = 3$. For the strongly magnetized plasmas we consider here, the particles are confined around the magnetic field lines with a high frequency. Averaging the full Vlasov equation over this cyclotron motion, which is faster than characteristic motions of interest, reduces dimensionality. The so-obtained equation, called gyrokinetic equation, describes the distribution function in a 5D phase space (3D in space and 2D in velocity).

The numerical solution of such Vlasov type equations is performed most of the time using Particle In Cell (PIC) methods where the plasma is approached by a finite number of macro-particles [1]. Even if these methods give satisfying results with relatively few particles, for some applications however, it is well known that the numerical noise inherent to the particle methods becomes too significant. Consequently, methods which discretize the Vlasov equation on a phase space grid have been proposed (see [2,3]) for plasma and beam physics applications. Among these Eulerian methods, we are interested in the implementation of the semi-Lagrangian method; it consists in updating the values of the distribution function at the nodes of the grid by following the characteristics ending at these nodes backward and interpolating the value at the bottom of the characteristics from the known values at the previous time step. Interpolation techniques that could be used are Lagrange, Hermite or spline for example.

The present work describes a parallel implementation of the semi-Lagrangian method by using a cubic spline interpolation method. Coupled with a time splitting procedure, the cubic spline interpolation seems to be a good compromise between accuracy and simplicity. Nevertheless, the standard method does not provide the locality of the reconstruction since all the values of the distribution function for a given 2D section are necessary to reconstruct $\bar{f}$ values in each cell of this 2D section. To overcome this problem of strong dependencies, we propose a solution that allows us to interpolate on quasi independent small 2D patches. Thus, we decompose global 2D sections into patches, each patch being devoted to a set of processor. One patch computes its own local cubic spline coefficients by solving reduced linear systems. Some adapted boundary conditions are imposed at the interface of the patches to obtain a $\mathcal{C}^1$ global solver which is close to the sequential solver that uses classical global splines. Moreover, thanks to a restrictive condition on the time step, the inter-processor communications are only done between logically adjacent sets of processors, which enables us to obtain competitive results from a communication cost point of view.

A first parallel simulator, that does not use the local interpolation method, was designed, but it does not scale well [4]. The present work focuses on the parallelization of a realistic semi-Lagrangian code which considers a full tokamak system. The rest of the paper is organized as follows. Section 2 focuses on the numerical scheme. Section 3 describes the sequential algorithms and Section 4 depicts the parallelization. Section 5 deals with performance analysis. These current researches are performed in an interdisciplinary approach together with physicists, mathematicians and computer scientists.

2 Numerical Scheme

The gyrokinetic model considers a distribution function $\bar{f}$ which depends on time and 5 other dimensions : r and θ are the polar coordinates in the shortest cross-section of the torus (called poloïdal section), φ refers to the angle in the largest cross-section of the torus, $v_\parallel$ is the velocity along the magnetic field lines (one has $v_\parallel = \frac{\mathrm{d}\varphi}{\mathrm{d}t}$), μ the magnetic moment corresponds to the action variable associated with the gyrophase (μ acts as a parameter because it is an adiabatic motion invariant). The time evolution of the guiding-center 5D gyroaveraged distribution function $\bar{f}_t(r, \theta, \varphi, v_\parallel, \mu)$ is governed by the gyrokinetic equation :

$$\frac{\partial \bar{f}}{\partial t} + \frac{\mathrm{d}r}{\mathrm{d}t}\frac{\partial \bar{f}}{\partial r} + \frac{\mathrm{d}\theta}{\mathrm{d}t}\frac{\partial \bar{f}}{\partial \theta} + \frac{\mathrm{d}\varphi}{\mathrm{d}t}\frac{\partial \bar{f}}{\partial \varphi} + \frac{\mathrm{d}v_\parallel}{\mathrm{d}t}\frac{\partial \bar{f}}{\partial v_\parallel} = 0. \tag{1}$$

$\bar{f}$ is periodic along θ and φ. Vanishing perturbations are imposed at the boundaries in the non-periodic directions, namely r and $v_\parallel$. $\bar{f}$ is initialized as an equilibrium distribution function $\bar{f}_{\mathrm{eq}}$ perturbed by a sum of accessible (m, n) Fourier modes (m and n being respectively the poloidal and toroidal wave numbers). That means, $\bar{f} = \bar{f}_{\mathrm{eq}}(1 + \delta\bar{f})$ where the perturbed part $\delta\bar{f} \propto \sum_{m,n} \cos(m\theta + n\varphi)$ and $\bar{f}_{\mathrm{eq}}(r,\mathcal{E}) = n_0(r) \times [2\pi T_i(r)/m_i]^{-\frac{3}{2}} \exp(-\mathcal{E}/T_i(r))$ with the energy $\mathcal{E} = \frac{1}{2} m_i v_\parallel^2 + \mu B(r,\theta)$. The radial temperature profile of the ions $T_i(r)$ (respectively of the electrons $T_e(r)$), the radial profile of density $n_0(r)$ and the magnetic field $B(r, \theta)$ are input data independant with respect to time.

The electric quasi-neutrality provides the self-consistency of the problem, coupling the electric potential $\Phi_t(r, \theta, \varphi)$ (which plays a major role in the $\frac{\mathrm{d}v_\parallel}{\mathrm{d}t}\frac{\partial \bar{f}}{\partial v_\parallel}$ term) to $\bar{f}$. The Φ function is found by solving the quasi-neutrality equation (denoting by $\nabla_\perp = (\partial_r, \frac{1}{r}\partial_\theta)$)

$$-\tfrac{1}{n_0(r)}\nabla_\perp \cdot \left[\tfrac{n_0(r)}{B_0\,\omega_c}\nabla_\perp \Phi\right] + \tfrac{e}{T_e(r)}\left[\Phi - \langle\Phi\rangle\right] = \bar{A}(r, \theta, \varphi) \tag{2}$$

$$\bar{A}(r, \theta, \varphi) = \tfrac{2\pi}{m_i\, n_0(r)} \int d\mu\, B(r,\theta) J_0(k_\perp \rho_c) \int dv_\parallel (\bar{f} - \bar{f}_{eq}) \tag{3}$$

The brackets $\langle\cdot\rangle$ refer to the magnetic flux surface average: $\langle\cdot\rangle = 1/(2\pi)^2 \iint \cdot\, \mathrm{d}\theta\, \mathrm{d}\varphi$. The Larmor radius corresponds to the notation ρ_c. The ion charge is $e_i = Z_i\, e$, and the ion mass m_i. The magnetic configuration is a circular concentric tokamak configuration with B_0 the value of magnetic field at the magnetic axis. The time is normalized to the inverse of the ion cyclotronic frequency $\omega_c = e_i\, B_0/m_i$. The zero-th order Bessel function J_0 corresponds to the gyroaverage operator in Fourier space. The variable $k_\perp$ is the transverse component of the wave vector.

Equations (1) and (2) are solved successively at each time step thanks to an explicit method. One deduces Φ_t from integral computations on $\bar{f}_t$ followed by the solution of equation (2). Then the electrostatic field $E_t(r, \theta, \phi) = -\nabla\Phi_t(r, \theta, \varphi)$ can be determined. The solution of (1) enables to update $\bar{f}_{t-dt}$ by $\bar{f}_{t+dt}$ using Φ_t, which yields second order accuracy in time.

3 Sequential Analysis

3.1 Global Algorithm

The Vlasov equation (1) is solved by splitting it into the advection equations:

$$\partial_t \bar{f} + \overrightarrow{v_{GC}} \cdot \overrightarrow{\nabla}_\perp \bar{f} = 0 \quad (\hat{r\theta} \text{ operator}),$$
$$\partial_t \bar{f} + v_\parallel \partial_\varphi \bar{f} = 0 \quad (\hat{\varphi} \text{ operator}), \qquad \partial_t \bar{f} + \hat{v_\parallel} \partial_{v_\parallel} \bar{f} = 0 \quad (\hat{v_\parallel} \text{ operator}).$$

where $\overrightarrow{v_{GC}} = \frac{\overrightarrow{E} \times \overrightarrow{B}}{B^2}$ is the drift velocity of the ion guiding center trajectories. Each advection consists in applying a shift operator. A Strang splitting procedure [2] is employed to reach second order accuracy. The sequence we choose is $(\hat{v_\parallel}, \hat{\varphi}, 2\,\hat{r\theta}, \hat{\varphi}, \hat{v_\parallel})$, where the factor 2 is a shift over an increased time step $2\,dt$.

At time step t, we will present key points of the different computations and their associated complexities. The algorithm manipulates two types of data structures: the 5D data $\bar{f}_{t-dt}, \bar{f}_t, \bar{f}_{t+dt}$, and the 3D data Φ and $\bar{A}$. The sizes of these structures are parametrized by the discretization along the different dimensions. Let N_r, N_θ, N_φ, $\mathrm{N}_{v_\parallel}$, N_μ be respectively the number of points in each dimension r, θ, φ, $v_\parallel$, μ. The 5D and 3D data size are $(\mathrm{N}_r\,\mathrm{N}_\theta\,\mathrm{N}_\varphi\,\mathrm{N}_{v_\parallel}\,\mathrm{N}_\mu)$ and $(\mathrm{N}_r\,\mathrm{N}_\theta\,\mathrm{N}_\varphi)$.

Algorithm 1: One time step in GYSELA

```
   // Vlasov solver
 1 for μ, r, θ in local subdomain do in //
 2 |  forall φ, v_∥ do
 3 |  |  1D splitting, operator v̂_∥
 4 |
 5 for μ, r, θ in local subdomain do in //
 6 |  forall φ, v_∥ do
 7 |  |  1D splitting, operator φ̂
 8 |
 9 for μ, r, θ in local subdomain do in //
10 |  forall φ, v_∥ do
11 |  |  2D splitting, parallel operator 2 r̂θ
12 |
13 for μ, r, θ in local subdomain do in //
14 |  forall φ, v_∥ do
15 |  |  1D splitting, operator φ̂
16 |
17 for μ, r, θ in local subdomain do in //
18 |  forall φ, v_∥ do
19 |  |  1D splitting, operator v̂_∥
20 |
21 |  // Field solver
22 Compute in parallel and broadcast Φ_{t+dt}
```

The Vlasov solver is composed of 5 splitting substeps. A substep requires the computation of the shift of each grid point, together with an interpolation step. The algorithmic complexity of each splitting substep is in $\Theta(\mathrm{N}_r\,\mathrm{N}_\theta\,\mathrm{N}_\varphi\,\mathrm{N}_{v_\parallel}\mathrm{N}_\mu)$. The field solver requires a traversal of data $\bar{f}_{t+dt}$ and the complexity is too in $\Theta(\mathrm{N}_r\,\mathrm{N}_\theta\,\mathrm{N}_\varphi\,\mathrm{N}_{v_\parallel}\mathrm{N}_\mu)$. Even if the Vlasov solver concentrates the major part of execution time, we have to consider the parallelization of every part to get an eventually scalable program. However, the parallelization of the field solver will not be described here.

4 Parallel Algorithm

4.1 Domain Decomposition

Concerning the Vlasov solver, the variable μ acts as a parameter. Then we give the responsibility of each value of μ to a given set of processors. Within such a set, a 2D domain decomposition allows us to attribute to each processor a subdomain

in (r,θ) dimensions. For a given local (μ,r,θ) tuple, a processor stores all values of $\bar{f}$ for $\varphi = *$ and $v_\parallel = *$. This data distribution leads to a straightforward parallelization of all parts of the Vlasov solver, excluding the $\hat{r\theta}$ operator part (see next subsection). The algorithm 1 introduces the computation distribution. Communications between processors are only required at lines 11, 22.

4.2 Local Spline Interpolation

In this section, we present an interpolation technique, based on a cubic spline method, in one dimension [5]. With a 2D tensor product of this spline method, interpolations on a 2D subdomain is achievable. In order to apply the $\hat{r\theta}$ operator, we use this 2D extension. Nevertheless, we explain here, only the 1D case to simplify the explanations.

Let us consider a function $\bar{f}$ which is defined on a global domain $[x_{\min}, x_{\mathrm{Max}}] \subset \mathbb{R}$. This domain is decomposed into several subdomains called generically $[x_{m_p}, x_{M_p-1}]$; each subdomain will be devoted to one processor p. In the following, we will use the notation $x_i = x_{m_p} + ih$, where h is the cell size: $h = (x_{M_p} - x_{m_p})/K$ and K the number of cells on a subdomain ($K \in \mathbb{N}$).

Let us now restrict the study of $f : x \mapsto f(x)$ on the interval $[x_{m_p}, x_{M_p}]$ with $M_p = m_p + K$. The projection s of f onto the cubic spline basis reads

$$f(x) \simeq s(x) = \textstyle\sum_{\nu=-1}^{K+1} \eta_\nu B_\nu(x),$$

where B_ν is the cubic B-spline. The interpolating spline s is uniquely determined by $(K+1)$ interpolating conditions and the Hermite boundary conditions at both ends of the interval in order to obtain a $\mathcal{C}^1$ global approximation

$$f(x_i) = s(x_i), \quad \forall i = m_p, ..., M_p, \quad f'(x_{m_p}) \simeq s'(x_{m_p}), \quad f'(x_{M_p}) \simeq s'(x_{M_p}). \tag{4}$$

The only cubic B-spline not vanishing at point x_i are $B_{i\pm1}(x_i) = 1/6$ and $B_i(x_i) = 2/3$. Hence (4) yields

$$f(x_i) = 1/6\ \eta_{i-1} + 2/3\ \eta_i + 1/6\ \eta_{i+1}, \quad i = m_p, \ldots, M_p. \tag{5}$$

On the other hand, we have $B'_{i\pm1}(x_i) = \pm 1/(2h)$, and $B'_i(x_i) = 0$. Thus the Hermite boundary conditions (4) become

$$f'(x_{m_p}) \simeq s'(x_{m_p}) = -\frac{1}{2h}\,\eta_{m_p-1} + \frac{1}{2h}\,\eta_{m_p+1}, f'(x_{M_p}) \simeq s'(x_{M_p}) = -\frac{1}{2h}\,\eta_{M_p-1} + \frac{1}{2h}\,\eta_{M_p+1}.$$

Finally, $\eta = (\eta_{m_p-1}, ...\eta_{M_p+1})^T$ is the solution of the $(K+3)\times(K+3)$ system $A\eta = F$, where F and A are

$$F = \left[f'(x_{m_p}), f(x_{m_p}), ..., f(x_{M_p}), f'(x_{M_p})\right]^T, \qquad A = \frac{1}{6}\begin{pmatrix} -3/h & 0 & 3/h & 0 & \cdots & 0 \\ 1 & 4 & 1 & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & -3/h & 0 & 3/h \end{pmatrix}.$$

A classical LU algorithm is used to solve the linear system $A\eta = F$ (known as the Thomas algorithm).

Approximation of the interface derivatives. In order to get accurate numerical simulations, one has to take care of the approximation of the derivatives at the interface of the subdomains. Various approximations were implemented. In order to recover the approximation of these interface derivatives obtained by a classical global cubic splines interpolation, a new formula can be derived to evaluate $f'(x_{m_p})$ and $f'(x_{M_p})$. Accurate numerical results are obtained with:

$$f'_{left}(x_i) = \sum_{j=1}^{j=10} \tilde{\gamma}_j^+ f_{i+j}\ ;\ f'_{right}(x_i) = \sum_{j=-10}^{j=-1} \tilde{\gamma}_j^- f_{i+j}\ ;\ f'(x_i) = f'_{left}(x_i) + f'_{right}(x_i).$$

We refer the reader to [5] for the details of the obtention of this approximation and values of coefficients γ_j^- and γ_j^+.

Properties. In our simulator we expect to interpolate on the interval $[x_{m_p-1}, x_{M_p}]$ instead of $[x_{m_p}, x_{M_p}]$. In order to extend the interpolation capability on one processor, we compute an extra η_{m_p-2} coefficient. We impose the property that $m_{p_i} = M_{p_j}$ for p_i and p_j two adjacent processors that share the grid point $x_{m_{p_i}}$. With these minor modifications, each processor has the responsibility to modify the values of grid points in the interval $[x_{m_p}, x_{M_p-1}]$, and have the capability to interpolate onto the extended interval $[x_{m_p-1}, x_{M_p}]$. In the 2D splitting phase, where the local splines are used, it means that the shift of one single grid point on the border of a subdomain must not exceed the elementary cell width. This constraint can be very restrictive and constitutes the main drawback of the method, since it is not possible to consider big shift (in r or θ) during a single advection in the 2D splitting phase. The computation of the η_{m_p-2} coefficient is deduced from (5) with $i = m_p - 1$. The term $f(x_{m_p-1})$ is received from a neighboring processor.

Communication pattern. In the parallel implementation of the local spline method, communications are required between adjacent processors to build the right hand side term F. On a local processor, the known values are $f(x_i)_{i\in[m_p,M_p-1]}$. For processors located at the borders of the global domain ($x_{\min} = x_{m_p}$ or $x_{\mathrm{Max}} = x_{M_p}$), boundary conditions (compact or periodic) are considered to retrieve the values of f needed outside the domain. Hereafter, we enumerate the data that lacks on the local processor to get F (excluding the specific problems that arise at the global domain boundaries):

1. Values of $f(x_{m_p-1})$, $f'_{left}(x_{m_p})$ are received from a neighboring processor.
2. Values of $f(x_{M_p})$, $f'_{right}(x_{M_p})$ are received from a neighboring processor.
3. The quantities $f'_{right}(x_{m_p})$ and $f'_{left}(x_{M_p})$ are computed on the local processor and send to processors that need them.

Concerning the item 3, we choose practically a large enough K to have only local calculations to compute $f'_{right}(x_{m_p})$ and $f'_{left}(x_{M_p})$. Experimentally, we have determined a lower bound on K: $K_{\min} = 32$, that leads to a relatively small overhead and provides good numerical stability.

In the case of a 2D interpolation, the F term is a matrix instead of a vector. The assembly of F requires communications with the 8 neighboring processors. On one processor and for a 2D patch of size $K_1 \times K_2$, the number of double precision real numbers to receive is $4\,(K_1 + K_2 + 4)$. This amount of communication could be compared to the interpolation cost of the $K_1 \times K_2$ points in a patch, which is in $\Theta(K_1\,K_2)$. For K_1 and K_2 greater to $K_{\min} = 32$, the ratio of communication cost over computation cost remains small.

Limitation. Numerical experiments with the local spline method for the 2D splitting on physical test cases have shown a bottleneck. The shifts in direction θ are often too large and above the limit we fixed (the width of one cell). It was not feasible to keep this configuration, so we were compelled to remove completely the θ parallelization in the algorithm (1). Another solution would have been to improve the interpolation capacity of each processor to larger subdomains. But in such case, extra unwanted communication would be required.

5 Performance Analysis

5.1 Hybrid Approach

The designed parallel simulator achieves good performance. Nevertheless, the limitation on the maximum number of processors that can be used, requires that we investigate other possible levels of parallelism. A refinement of the MPI parallelization would require a fair amount of code restructuring and would imply new communication schemes. Indeed, the MPI and OpenMP programming models can be combined into a hybrid paradigm to exploit levels of parallelism at a finer grain, without heavy code manipulation. The hybrid approach is suitable for clusters of SMP nodes where MPI provides communication capability across nodes and OpenMP exploits loop level parallelism within a node.

We add several parallel loops in all parts of the algorithm 1. A parallelization in variable φ is adequate for the 1D splitting in $v_\parallel$, the 2D splitting in (r, θ). A parallel loop in θ allows a simple formulation in the 1D splitting in φ. The field solver uses too OpenMP parallelization.

5.2 Efficiency of the Parallelization

Numerical experiments were performed on a cluster of IBM 16-core nodes located at Bordeaux, France. Each node hosts Power5 processors and offers 27GB of shared memory.

Hereafter, we present a 4D test case with $\mathrm{N}_\mu = 1$. The variable μ corresponds to the coarser level of parallelism. So, if we imagine running the same test case in a 5D configuration (with $\mathrm{N}_\mu = 16$) on 16 times more processors, we should observe quite similar scalability performances for advections.

Let us recall that we are limited down by width $K_{\min} = 32$ for 2D patches (see subsection 4.2), and the MPI parallelization on variable θ is not active; so in the

Table 1. Efficiency and computation time in seconds for a single time step of a medium 4D test case with hybrid OpenMP/MPI solution $N_r = 256, N_\theta = 256, N_\varphi = 128, N_{v_\parallel} = 64, N_{\mu=1}$ (nbt the number of threads within each MPI process, and [*Nb. procs*/nbt] the number of MPI processes)

	Time	Effic.	Time	Effic.	Time	Effic.	Time	Effic.
Nb. processors	**1** ($nbt=1$)		**8** ($nbt=1$)		**64** ($nbt=8$)		**128** ($nbt=16$)	
advections 1D (φ)	334.7	100	40.87	102	5.38	99	2.66	98
advections 1D ($v_\parallel$)	305.6	100	40.22	95	5.04	95	2.52	95
advection 2D (r, θ)	709.5	100	99.67	89	12.94	85	6.86	81
Total advections	1349.8	100	180.76	93	23.37	90	12.04	88
Total field sover	33.1	100	9.95	42	1.50	33	0.78	33

given test case with $N_r = 256$, the (r, θ) domain could be decomposed up to only $proc_r = 8$ subdomains. The maximum number of processors that we could use is then $proc_r\ N_\mu = 8$, which is a small number for a 4D test case requiring large computation time. The hybrid paradigm usage increases this maximum number to 128 (one could use up to 16 threads per node), thus improving scalability.

In table 1, the 1D splittings are perfectly parallel and scalable, because no overhead in computation nor in communication is needed. However, the 2D splitting requires a communication step to transmit boundary coefficients and derivatives. Furthermore, the 2D interpolation on patches induces a small computation overhead in comparison to a global spline sequential method. These two facts explain why the efficiency decays whenever $proc_r$ and number of processors increases from 1 to 8. The field solver, because of a too much simple parallelization has not a good speedup. Nevertheless, computation time for this field solver remains small compared to others.

The main advantages of the hybrid approach is to allow one to use more processors for a given test case. These results demonstrated the overall scalability of the application.

6 Conclusion

We describe the parallelization of a numerical simulator that solves a 5D Vlasov system[1]. The scalability is really impressive on a cluster of SMP nodes thanks to good properties of the local spline method. Furthermore, multiple levels of parallelism are used by combining message passing and OpenMP parallelization. The hybrid approach leads to an application that could use more processors than in a MPI-only approach for a given test case size. The scalability improvement will enable us to run the code with good efficiency on hundreds of processors.

[1] Acknowledgments: This work was partially suported by EURATOM/CEA, contract V.3529.001.

References

1. Birdsall, C., Langdon, A.: Plasma Physics via Computer Simulation. Institute of Physics Publishing, Bristol and Philadelphia (1991)
2. Cheng, C., Knorr, G.: The integration of the Vlasov equation in configuration space. J. Comput Phys. 22, 330 (1976)
3. Filbet, F., Sonnendrücker, E., Bertrand, P.: Conservative numerical schemes for the Vlasov equation. J. Comput. Phys. 172(1), 166–187 (2001)
4. Grandgirard, V., Brunetti, M., Bertrand, P., Besse, N., Garbet, X., Ghendrih, P., Manfredi, G., Sarazin, Y., Sauter, O., Sonnendrücker, E., Vaclavik, J., Villard, L.: A drift-kinetic semi-Lagrangian code for ion turbulence simulation. J. Comput. Phys. 217, 395–423 (2006)
5. Crouseilles, N., Latu, G., Sonnendrücker, E.: Hermite spline interpolation on patches for a parallel solving of the Vlasov-Poisson equation. Technical Report 5926, Research report INRIA (2006), http://hal.inria.fr/inria-00078455/en/

Automatic Parallelization of Object Oriented Models Executed with Inline Solvers

Håkan Lundvall and Peter Fritzson

PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{haklu,petfr}@ida.liu.se

Abstract. In this work we report preliminary results of automatically generating parallel code from equation-based models together at two levels: Performing inline expansion of a Runge-Kutta solver combined with fine-grained automatic parallelization of the resulting RHS opens up new possibilities for generating high performance code, which is becoming increasingly relevant when multi-core computers are becoming common-place. We have introduced a new way of scheduling the task graph generated from the simulation problem which utilizes knowledge about locality of the simulation problem.

Keywords: Modelica, automatic parallelization.

1 Background – Introduction to Mathematical Modeling and Modelica

Modelica is a rather new language for equation-based object-oriented mathematical modeling which is being developed through an international effort [4], [5]. The language unifies and generalizes previous object-oriented modeling languages. Modelica is intended to become a *de facto* standard. It allows defining simulation models in a declarative manner, modularly and hierarchically and combining various formalisms expressible in the more general Modelica formalism. The multidomain capability of Modelica gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model.

In the context of Modelica *class libraries* software components are Modelica classes. However, when building particular models, components are *instances* of those Modelica classes. Classes should have well-defined communication interfaces, sometimes called ports, in Modelica called *connectors*, for communication between a component and the outside world. A component class should be defined *independently of the environment* where it is used, which is essential for its *reusability*. This means that in the definition of the component including its equations, only local variables and connector variables can be used. No means of communication between a component and the rest of the system, apart from going via a connector, is allowed. A component may internally consist of other connected components, i.e. *hierarchical* modeling.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 365–372, 2007.

To grasp this complexity a pictorial representation of components and connections is quite important. Such graphic representation is available as *connection diagrams*.

To summarize, Modelica has improvements in several important areas:

- *Object-oriented mathematical modeling*. This technique makes it possible to create physically relevant and easy-to-use model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.
- *Acausal modeling*. Modeling is based on *equations* instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases re-usability of model components, since components adapt to the data flow context in which they are used. This generalization enables both simpler models and more efficient simulation.
- *Physical modeling of multiple application domains*. Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks with fixed input/output causality.

2 Approaches to Integrate Parallelism and Mathematical Models

There are several approaches to exploit parallelism in mathematical models. In this section we briefly review some approaches that are being investigated in the context of parallel simulation of Modelica models.

2.1 Automatic Parallelization of Mathematical Models

One obstacle to parallelization of traditional computational codes is the prevalence of low-level implementation details in such codes, which also makes automatic parallelization hard.

Instead, it would be attractive to directly extract parallelism from the high-level mathematical model, or from the numerical method(s) used for solving the problem. Such parallelism from mathematical models can be categorized into three groups:

- *Parallelism over the method*. One approach is to adapt the numerical solver for parallel computation, i.e., to exploit parallelism over the method. For example, by using a parallel ordinary differential equation (ODE) solver for that allows computation of several time steps simultaneously. However, at least for ODE solvers, limited parallelism is available. Also, the numerical stability can decrease by such parallelization.
- *Parallelism over time*. A second alternative is to parallelize the simulation over the simulated time. This is however best suited for discrete event simulations, since solutions to continuous time dependent equation systems develop sequentially over time, where each new solution step depends on the immediately preceding steps.
- *Parallelism of the system*. This means that the modeled system (the model equations) are parallelized. For an ODE or DAE equation system, this means parallelization of the right-hand sides of such equation systems which are available

in explicit form; moreover, in many cases implicit equations can automatically be symbolically transformed into explicit form.

A thorough investigation of the third approach, automatic parallelization over the system, has been done in our recent work on automatic parallelization (fine-grained task-scheduling) of a mathematical model [1],[10]. Speedup measurements from this investigation can be seen in figure 2.

In this work we aim at extending our previous approach to inlined solvers, integrated in a framework exploiting several levels of parallelism.

2.2 Coarse-Grained Explicit Parallelization Using Computational Components

Automatic parallelization methods have their limits. A natural idea for improved performance is to structure the application into computational components using strongly-typed communication interfaces.

This involves generalization of the architectural language properties of Modelica, currently supporting components and strongly typed connectors, to distributed components and connectors. This will enable flexible configuration and connection of software components on multiprocessors or on the GRID. This only involves a structured system of distributed solvers/ or solver components.

2.3 Explicit Parallel Programming

The third approach is providing general easy-to-use explicit parallel programming constructs within the algorithmic part of the modeling language. We have previously explored this approach with the NestStep-Modelica language [6], [11].

3 Combining Parallelization at Several Levels

Models described in object oriented equations based languages like Modelica render large differential algebraic equation systems that can be solved using numerical ODE-solvers. Many scientific and engineering problems require a lot of computational resources, particularly if the system is large or if the right hand side is complicated and expensive to evaluate. Obviously, the ability to parallelize such models is important, if such problems are to be solved in a reasonable amount of time.

As mentioned in Section 2, parallelization of object oriented equation based simulation code can be done at several different levels. In this paper we explore the combination of the following two parallelization approaches:

- Parallelization across the method, e.g., where the stage vectors of a Runge-Kutta solver can be evaluated in parallel within a single time step
- Fine grained parallelization across the system where the evaluation of the right hand side of the system equations is parallelized.

In previous work [1] automatic parallelization across the system has been done by building a task graph containing all the operations involved in evaluating the equations of the system DAE. In order to make the cost of evaluating each task large

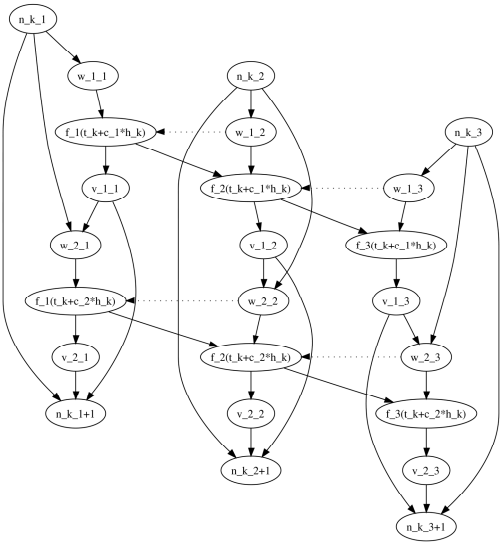


Fig. 1. Task graph of two stage inlined Runge-Kutta solver

enough compared to the communication cost between the parallel processors he uses a graph rewriting system that merges tasks together in such a way that the total cost of computing and communicating is minimized. The solver is centralized and runs on one processor. Each time the right hand side is to be evaluated, data needed by tasks on other processors is send and the result of all tasks is collected in the first process before returning to the solver. As a continuation of this work we now inline an entire Runge-Kutta solver in the task graph before scheduling of the tasks.

Many simulation problems have DAE:s consisting of a very large set of equations but were each equation only depends on a relatively small set of other equations.

Let $f = (f_1,\ldots,f_n)$ be the right hand side of such a simulation problem and let f_i contain equations only depending on equations of components of indices in a range near i. This makes it possible to pipeline the computations of the resulting task graph, since evaluating f_i for stage s of the Runge-Kutta solver depend only on f_j of stage s for j close to i and on f_i of stage $s-1$.

A task graph of a system where the right hand side can be divided into three parts, denoted by the functions f_1, f_2 and f_3 where f_i only depend on f_{i-1}, inlined in a two stage Runge-Kutta solver is shown in figure 1. In the figure n_k represent the state after the previous time step. We call the function f_i the blocks of the system. If we schedule each block to a different processor, let us say f_i is scheduled to p_i, then p_1 can continue calculating the second stage of the solver as p_2 starts calculating the first stage of f_2. The communication between p_1 and p_2 can be non-blocking so that if many stages are used communication can be carried out simultaneous to the calculations.

The pipelining technique is described in [9]. Here we aim to automatically detect pipelining possibilities in the total task graph containing both the solver stages and the right hand side of the system, and automatically generate parallelized code optimized for the specific latency and bandwidth parameters of the target machine.

It the earlier approach with task merging including task duplication the resulting task graph usually ends up with one task per processor and communication takes place at two points in each simulation step; initially when distributing the previous step result from the processor running the solver to all other processors and at the end collecting the results back to the solver.

When inlining a multi-stage solver in the task graph each processor only needs to communicate whit its neighbor. In this approach however we cannot merge tasks as much since the neighbors of a processor depends on initial results to be able to start their tasks. So, instead of communicating a lot in the beginning and in the end smaller portions are communicated throughout the calculation of the simulation step.

If the task graph of a system mostly has the property of having a narrow access distance, which is required for the pipelining, but only on a small number of places access components in more distant parts of the graph.

4 Pipelining the Task Graph

Since communication between processors is going to be more frequent with this approach we want to make sure the communication interfere as little as possible with computation. Therefore, we schedule the tasks in such a way that communication taking place inside the simulation step is always directed from a processor with lower rank to a higher ranked processor. In this way the lower ranked processor is always able to carry on with calculations even if the receiving processor temporarily falls behind. At the end of the simulation step there is a face were values required for the next simulation step is transferred back to lower ranked processors, but this is only needed once per simulations step instead of once for each evaluation of the right hand side. Further more this communication takes place between neighbors and not to a single master process which otherwise can get overloaded with communication as the number of processors becomes large.

4.1 Sorting Equations for Short Access Distance

One part of translating an acausal equation-based model into simulation code involves sorting the equations into data dependency order. This is done using Tarjan's algorithm which also finds any strongly connected components in the system graph, i.e., a group of equations that must be solved simultaneously. We assign a sequence number to each variable, or set of variables in case of a strongly connected component, and use this to help the scheduler assign tasks that communicate much within the same processor. When the task graph is generated each task is marked with sequence number of the variable it calculates. When a system with n variables is to be scheduled onto p processors, tasks marked 1 through n/p is assigned to the first processor and so on.

Even though Tarjan's algorithm assures that the equations are evaluated in a correct order we cannot be sure that there is not a different ordering where the access distance is smaller. If for example two parts of the system is largely independent they can become interleaved in the sequence of equations making the access distance unnecessarily large. Therefore we apply an extra sorting step after Tarjan's algorithm

which moves equations with direct dependencies closer together. This reduces the risk of two tasks with a direct dependency getting assigned to different processors.

As input to the extra sorting step we have a list of components and a matching defining which variable is solved by which equation. On component represent a set of equations that must be solved simultaneously. A component often includes only one equation. The extra sorting step works by popping a component from the head of the component list and placing them in the resulting sorted list as near the head of the sorted list as possible without placing it before a component on which it depends.

4.2 Scheduling

In this section we describe the scheduling process. We want all communication occurring inside the simulation step to be one-way only, from processors with lower rank to processors with higher rank. To achieve this we make use of information stored with each task telling us from which equation it originates and thus which variable it is a part of evaluating. We do this by assigning the tasks to the processors in the order obtained after the sorting step described in section 6.

Task with variable number *0* through n_1 is scheduled to the first processor, n_1+1 through n_2 to the second and so on. The values of n_i are chosen so that they are always the variable number representing a state variable.

If we generate code for a single stage solver, e.g., Euler, this would be enough to ensure backward communication only takes place between simulation steps, since the tasks are sorted to ensure no backward dependencies. This is not, however, the case when we generate code for multi-stage solvers. When sorting the equations in data-dependency order, variables considered known, like the state of the previous step are not considered, but in a later stage of the solver those values might have been calculated by an equation that comes later in the data-dependency sorting. This kind of dependency is represented by the dotted lines in figure 1. Luckily such references tend to have a short access distance as well and we solve this by adding a second step to the scheduling process.

For each processor p starting with the lowest ranked, find each task reachable from any leaf task scheduled to p by traversing the task graph with the edges reversed. Any task visited that was not already assigned to processor p is then moved to processor p. Tests show that the moved tasks do not influence the load balance of the schedule much.

5 Measurements

In order to evaluate the gained speedup we have used a model of a flexible shaft using a one-dimensional discretization scheme. The shaft is modeled using a series of n rotational spring-damper components connected in a sequence. In order to make the simulation task computationally expensive enough, to make parallelization worth while, we use a non linear spring-damper model. In these tests we use a shaft consisting of 100 spring-damper elements connected together. The same model has

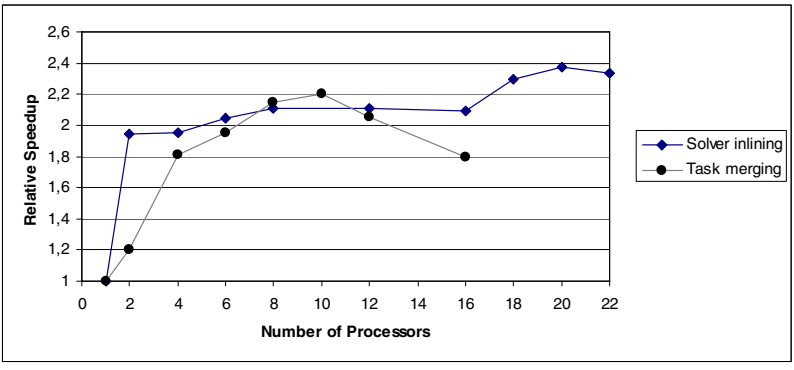


Fig. 2. Relative speedup on Linux-cluster. The new technique compared to the previous task merging technique.

bee used when the task merging approach was evaluated in [1], which makes it possible to compare the results of this work to what was previously achieved.

The measurements were carried out on a 30-node PC cluster where each computation node is equipped with two 1.8 GHz AMD Athlon MP 2200+ and 2GB of RAM. Gigabit Ethernet is used for communication.

Figure 2 shows the results of the tests carried out so far. As can be seen the speedup for two processors is almost linear, but when the number of processors increase the speedup does not follow.

6 Conclusion and Future Work

To conclude we can se that for two processors the tests were very promising, but those promises were not fulfilled when the number of processors increased. If we compare to the previous results obtained with task merging in [1], though, we do not suffer from slowdown in the same way (see figure 2). Most likely this has to do with the fact that the communication cost for the master process running the solver increases linearly with the number of processors whereas in our new approach this communication is distributed more evenly among all processors.

In the nearest future we will profile the generated code to see were the bottlenecks are when ran on more than two processors and see if the scheduling algorithm can be tuned to avoid them. Also, tests must be carried out on different simulation problems to see if the results are general or if it differs much depending on the problem.

We also intend to port the runtime to run on threads in a shared memory setup. Since the trend is for CPU manufacturers to add more and more cores to the CPUs, it is becoming more and more relevant to explore parallelism in such environments.

A runtime for the Cell BE processor is also planed. This processor has eight, so called, Synergistic Processing Elements (SPE) which their own local memory. Transfers to and from those local memories can be carried out using DMA without using any computation resources.

Acknowledgements

This work was supported by Vinnova in the Safe & Secure Modeling and Simulation project.

References

1. Aronsson, P.: Automatic Parallelization of Equation-Based Simulation Programs. PhD thesis, Dissertation No 1022, Dept. Computer and Information Science, Linköping University, Linköping, Sweden
2. Bonorden, O., Juurlink, B., von Otte, I., Rieping, I.: The Paderborn University BSP (PUB) Library. Parallel Computing 29, 187–207 (2003)
3. Fritzson, P., Aronsson, P., Lundvall, H., Nyström, K., Pop, A., Saldamli, L., Broman, D.: The OpenModelica Modeling, Simulation, and Software Development Environment. In: Simulation News Europe, 44/45, (December 2005) See also, http://www.ida.liu.se/projects/OpenModelica
4. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, p. 940 ISBN 0-471-471631, Wiley-IEEE Press (2004) See also book web page http://www.mathcore.com/drModelica
5. The Modelica Association. The Modelica Language Specification Version 2.2 (March 2005), http://www.modelica.org
6. Sohl, J.: A Scalable Run-time System for NestStep on Cluster Supercomputers. Master thesis LITH-IDA-EX-06/011-SE, IDA, Linköpings universitet, 58183 Linköping, Sweden (March 2006)
7. Nyström, K., Fritzson, P.: Parallel Simulation with Transmission Lines in Modelica. In: Proceedings of the 5th International Modelica Conference (Modelica'2006), Vienna, Austria (September 4-5, 2006)
8. Siemers, A., Fritzson, D., Fritzson, P.: Meta-Modeling for Multi-Physics Co-Simulations applied for OpenModelica. In: Proceedings of International Congress on Methodologies for Emerging Technologies in Automation (ANIPLA2006), Rome, Italy (November 13-15, 2006)
9. Korch, M., Rauber, T.: Optimizing Locality and Scalability of Embedded Runge-Kutta Solvers Using Block-Based Pipelining. Journal of Parallel and Distributed Computing 66(3), 444–468 (2006)
10. Aronsson, P., Fritzson, P.: Automatic Parallelization in OpenModelica. In: Proceedings of 5th EUROSIM Congress on Modeling and Simulation, Paris, France. ISBN (CD-ROM) 3-901608-28-1 (September 2004)
11. Kessler, C., Fritzson, P., Eriksson, M.: NestStepModelica: Mathematical Modeling and Bulk-Synchronous Parallel Simulation. PARA-06 Workshop on state-of-the-art in scientific and parallel computing, Umeå, Sweden (June 18-21, 2006)

3D Parallel Elastodynamic Modeling of Large Subduction Earthquakes

Eduardo Cabrera[1], Mario Chavez[2,3], Raúl Madariaga[3], Narciso Perea[2], and Marco Frisenda[3]

[1] Supercomputing Dept., DGSCA, UNAM, C.U., 04510, Mexico DF, Mexico
[2] Institute of Engineering, UNAM, C.U., 04510, Mexico DF, Mexico
[3] Laboratoire de Géologie CNRS-ENS, 24 Rue Lhomond, Paris, France
eccf@super.unam.mx, marioch48@hotmail.com, raul.madariaga@ens.fr, narpere@hotmail.com, mfrisenda@yahoo.it

Abstract. The 3D finite difference modeling of the wave propagation of M>8 earthquakes in subduction zones in a realistic-size earth is very computationally intensive task. We use a parallel finite difference code that uses second order operators in time and fourth order differences in space on a staggered grid. We develop an efficient parallel program using message passing interface (MPI) and a kinematic earthquake rupture process. We achieve an efficiency of 94% with 128 (and 85% extrapolating to 1,024) processors on a dual core platform. Satisfactory results for a large subduction earthquake that occurred in Mexico in 1985 are given.

Keywords: Elastodynamic, modeling, earthquakes, parallel computing.

1 Introduction

The 19/09/1985 a large Ms 8.1 subduction earthquake occurred on the Mexican Pacific coast with an epicenter at about 340 km from Mexico City is shown in Fig. 1A. The rupture area of this event of about 180 x 100 km is also shown in this figure. In Fig. 1B, a profile from the Mexican coast and beyond Mexico City shows the tectonic plates involved in the generation of this type of earthquakes in Mexico. Finally, the kinematic representation of the average slip associated to the mentioned earthquake is presented in Fig. 1C. As the recurrence time estimated for this highly destructive type of events in Mexico is of only a few decades, there is a seismological and engineering interest in modeling them [1].

Herewith, we developed an efficient parallel program using message passing interface (MPI) with a kinematic specification of the rupture process in the fault. In Ch. 2 we synthesize the elastodynamics of the problem; the data parallelism approach decomposition proposed and the MPI implementation are presented in Ch. 3. The study of the efficiency of the proposed parallel program is discussed in Ch. 4 and in Ch. 5 results obtained for the modeling of the seismic wave propagation of the 19/09/1985 Ms 8.1 subduction earthquake are given.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 373–380, 2007.

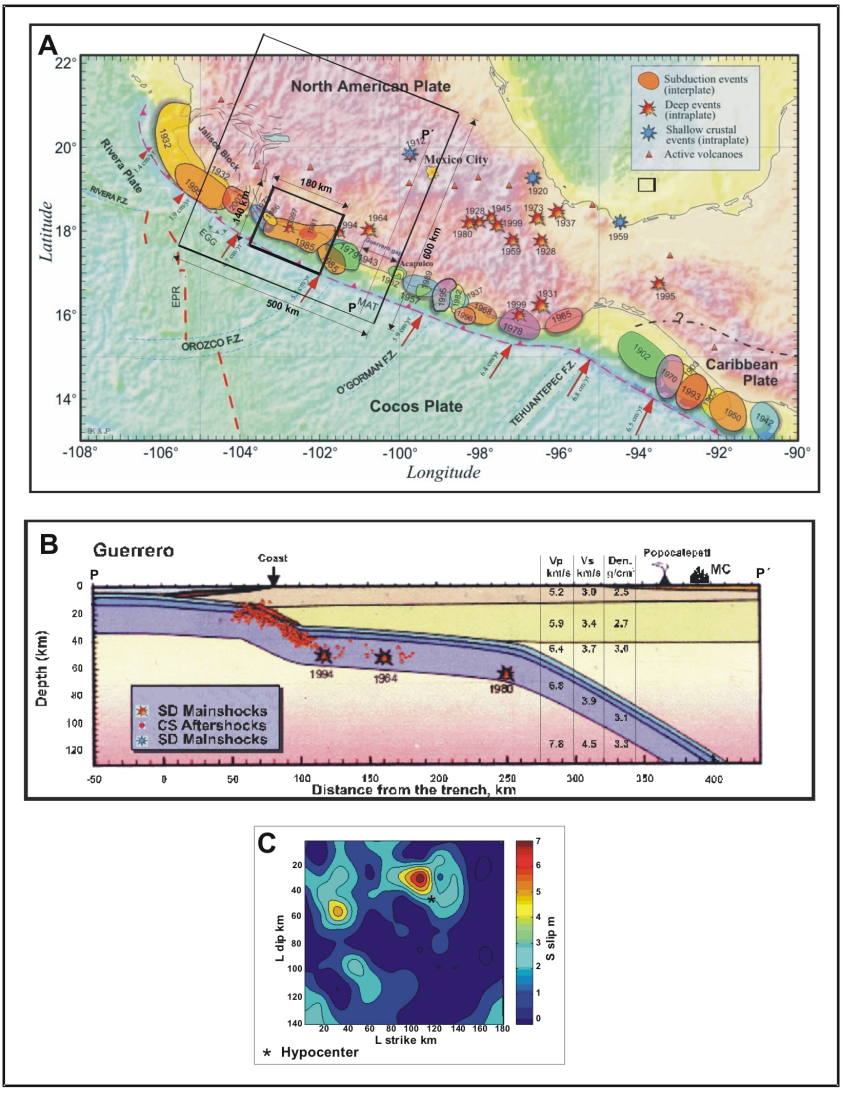


Fig. 1. A) Inner rectangle is the rupture area of the 19/09/1985 Ms 8.1 earthquake on surface projection of the 500x600x124 km earth crust volume 3DFD discretization; B) profile P-P´; C) Kinematic slip distribution of the rupture of the 1985 earthquake [4]

2 Elastodynamics and the 3DFD Algorithm

A synthesis of the elastodynamic formulation and its algorithm description of the elastic wave propagation problem are presented by following [2]. The elastic wave equation in a 3D medium occupying a volume V and boundary S, the medium may be described using Lamé parameters $\lambda(\overline{x})$ and $\mu(\overline{x})$ and mass density $\rho(\overline{x})$, where

$(\overline{x}) \in \Re^3$. The velocity-stress form of the elastic wave equation consists of nine coupled, first order partial differential equations for the three particle velocity vector components $v_{ij}(\overline{x},t)$ and the six independent stress tensor components $\sigma_{ij}(\overline{x},t)$, where i,j = 1,2,3 and assuming that $\sigma_{ij}(\overline{x},t) = \sigma_{ji}(\overline{x},t)$:

$$\frac{\partial v_i(\overline{x},t)}{\partial t} - b(\overline{x})\frac{\partial \sigma_{ij}(\overline{x},t)}{\partial x_j} = b(\overline{x})\left[f_i(\overline{x},t) + \frac{\partial m_{ij}^a(\overline{x},t)}{\partial x_j} \right]. \quad (1)$$

$$\frac{\partial \sigma_{ij}(\overline{x},t)}{\partial t} - \lambda(\overline{x})\frac{\partial v_k(\overline{x},t)}{\partial x_k}\delta_{ij} - \mu(\overline{x})\left[\frac{\partial v_i(\overline{x},t)}{\partial x_j} + \frac{\partial v_j(\overline{x},t)}{\partial x_i} \right] = \frac{\partial m_{ij}^s(\overline{x},t)}{\partial t} \quad (2)$$

where $b=1/\rho$, and f_i is the force source tensor and $m_{ij}^a(\overline{x},t) = 1/2\left[m_{ij}(\overline{x},t) - m_{ji}(\overline{x},t)\right]$, $m_{ij}^s(\overline{x},t) = 1/2\left[m_{ij}(\overline{x},t) + m_{ji}(\overline{x},t)\right]$ are the moment antisymmetric and symmetric source tensors and δ_{ij} is Dirac´s δ. The traction boundary condition (normal component of stress) must satisfy

$$\sigma_{ij}(\overline{x},t) n_j(\overline{x}) = t_i(\overline{x},t) \ . \quad (3)$$

for $\overline{x}$ on *S*, where $t_i(\overline{x},t)$ are the components of the time-varying surface traction vector and $n_i(\overline{x})$ are the components of the outward unit normal to *S*. The initial conditions on the dependent variables are specified at V and on S at time $t = t_0$ by

$$v_i(\overline{x},t) = v_i^0(\overline{x}), \ \sigma_{ij}(\overline{x},t) = \sigma_{ij}^0(\overline{x}). \quad (4)$$

On output, the code produces both seismograms and 2D plane slices. If the orientation of interest is on a particular axis defined by the dimensionless unit vector $\overline{b}$, then the particle velocity seismogram is:

$$v_b(\overline{x}_r,t) = b_k v_k(\overline{x}_r,t) = b_1 v_1(\overline{x}_r,t) + b_2 v_2(\overline{x}_r,t) + b_3 v_3(\overline{x}_r,t). \quad (5)$$

Details about the staggered finite difference scheme on which the algorithm used is based can be found in [3].

3 Parallel Implementation

We use data parallelism for efficiency. The best parallel programs are those where each processor gets almost the same amount of work while trying to minimize communications. Using this kind of partition, the domain is decomposed into small pieces (subdomains) and distributed among all processors; therefore, each processor solves its own subdomain problems. 3D domain decomposition is shown in Fig. 2.

For the process discussed in this paper, 1D, 2D, and 3D decomposition are possible; however, we encourage the 3D one because it is well-balanced, extremely efficient and the more appropriate for the elastic wave propagation code as large problems –too big to fit on a single processor.

We use message passing interface (MPI) to parallelize 3DFD. The fourth order spatial finite difference scheme requires two additional planes of memory on every face of the subdomain to compute properly the finite difference solution independently from the other processes; therefore, we allocate padded subdomains of memory for every face of the subdomain cube (shown in the bottom of Fig. 2) to assure the precise functioning of the staggered finite difference scheme used.

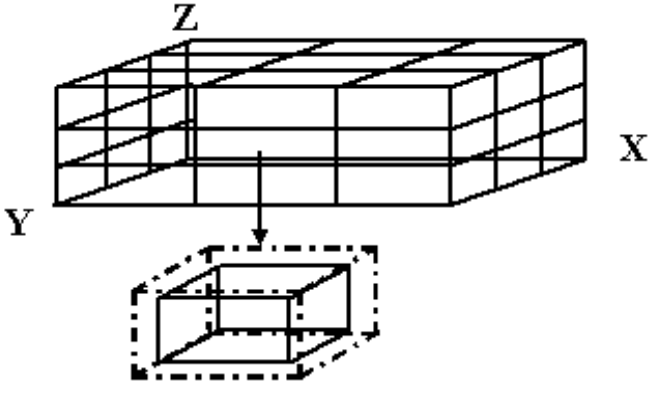


Fig. 2. 3D decomposition using data parallelism and an independent subdomain with ghost cells as dashed lines

Parallel I/O is used in the program that allows us to model a large realistic-size model. The input basic run parameters and geometry data which are scalars are read and broadcast by processor 0. The earth model data is read by all processors using collective I/O. The output part of the program uses, as well, collective I/O to write plane slices and seismograms. We do not measure the time spent in such phases because it is in the time-step loop where the majority of the time is spent. We use MPI *shift* commands to communicate neighboring's edges.

4 Efficiency

Speedup, Sp, and efficiency, E, among others, are the most important metrics to characterize the performance of parallel programs. Theoretically, speedup is limited by Amdahl's law [5]. For a scaled-size problem, one must estimate the running time on a single processor [2]. Sp and E are defined as

$$Sp \equiv \frac{mT_1(n/m)}{T_m(n)}, \qquad E \equiv \frac{T_1(n/p)}{mT_m(n)}. \tag{6}$$

where T_1 is the serial time execution and T_m is the parallel time execution on m processors for a size problem n.

We can estimate the cost for this parallel algorithm straightforward without I/O timings because the largeness of the work occurs in the elastic wave propagation. Therefore, we must estimate computation and communication terms

$$T(n,m) = \tau_{comp}(n,m) + \tau_{comm}(n,m) \quad . \tag{7}$$

where τ_{comp} is the computation cost and τ_{comm} represent the communication cost on m processors for a size problem n.

There are two main machine constants which most impact the speed of message communication that are *bandwidth,* β, -message dependant- which is (the reciprocal of) transmission time/byte, and latency, ι, represents the startup cost of sending an message -independent of message size. Therefore, the cost to send a single message with χ length of data is $\iota + \chi\beta$.

We use 128 processors of UNAM HP Cluster Platform 4000, which has Opteron dual core processors (1,368 cores) of 2.6GHz with Infiniband interconnection (known in short as KanBalam [6]). KanBalam has the following time constants: $\beta = 1\times10^{-9}$, $\iota = 13\times10^{-6}$, and the computation time per flop, $\Gamma = 1.9\times10^{-13}$, all of them are in seconds. The size of each subdomain is $Nx \times Ny \times Nz$, that we call R for simplicity, where Nx=500, 1000, 2000, 4000; Ny=600, 1200, 2400, 4800 and Nz=124, 248, 496, 992 are model size per direction; therefore, the cost of performing a finite difference calculation on $npx \times npy \times npz$ *,m,* processors is $A\Gamma R^3 / m$, where A is the number of floating operations in the finite difference scheme (velocity-stress consists of nine coupled variables). As we stated above, this scheme requires us to communicate two neighboring's planes in the 3D decomposition plus four extra planes for cubic extrapolation –necessary if the user specifies a receiver or slice plane not on a grid node; therefore, communication costs for a 1D decomposition are –at most- $8(\iota + 4\beta R^2)$, where the factor 4 is the size in bytes of memory of each data grid. $16(\iota + 4\beta R^2 / \sqrt{m})$ is the cost for a 2D decomposition, and for a 3D decomposition we have $24(\iota + 4\beta R^2 / m^{2/3})$. In short, for a 3D decomposition we have the following

$$T(n,m) = A\Gamma R^3 / m + 24(\iota + 4\beta R^2 / m^{2/3}) \, . \tag{8}$$

and

$$Sp \equiv \frac{A\Gamma R^3}{A\Gamma R^3 / m + 24(\iota + 4\beta R^2 / m^{2/3})} \, . \tag{9}$$

The communication cost depends on both the order the finite difference scheme and the type of processor decomposition.

Results for different size models and number of processors (P) from 1-1,024 are shown in Table 1 and Fig. 3. I/O timings are not reported.

Table 1. Scaled-sized model: processors used in each axis, timings, speedup, efficiency and memory per subdomain (mps) obtained (The 1,024 processors results are based on (8) and (9))

Size model and spatial step (dh, km)	P	Px	Py	Pz	Total run time (s)	Speedup (Sp)	Efficiency (E)	Mps (GB)
500x600x124 (1)	1	1	1	1	34187.7	1	1	2.08
1000x1200x248 (0.5)	16	1	4	4	33201.5	16.47	1.03	1.042
2000x2400x496 (0.25)	128	4	8	4	36230.3	120.8	0.94	1.042
4000x4800x992 (0.125)	*1024*	*16*	*16*	*4*	*39986.3*	*876*	*0.85*	*1.042*

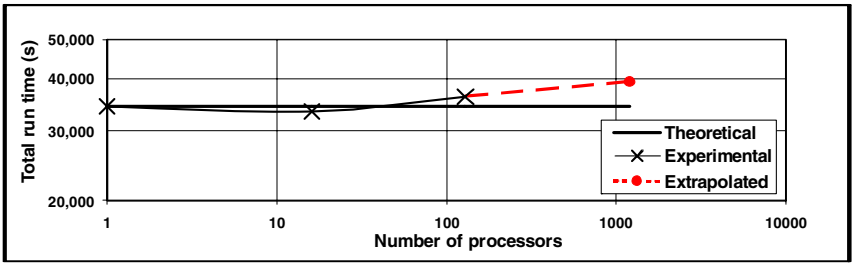


Fig. 3. Running time for the models run on KanBalam. (The 1,024 processors results are based on (8) and (9)).

5 Results for the 19/09/1985 Mexico's Ms 8.1 Subduction Earthquake

Herewith we present two examples of the type of results that were obtained with the 3D parallel MPI code implemented: the low frequency velocity field patterns in the X direction, Fig. 2 and the seismograms obtained at observational points in the so-called near and far fields of the wave propagation pattern.

Three spatial discretizations of the earth crust volume were used: dh = 1, 0.5, and 0.25 km. In Fig. 4A, we present two snapshots of the wave propagation patterns in the X direction obtained 48 and 120s after the initiation of the kinematic rupture of the seismic source, They correspond to the dh = 0.5 km discretization, notice in Fig. 4A that at 48s the main seismic effects are occurring in the near field, i.e. on top of the seismic source, while the opposite is observed at 120s, when the seismic waves are fully developed in the far field, where Mexico City is located with respect to the source.

In Fig. 4B the synthetic seismograms obtained for dh = 1, 0.5 and 0.25 km, at an observation site practically on top of the largest "subevent" of the 1985 Mexico earthquake (Fig. 1C) are presented, Notice in Fig. 4B, that, the maximum amplitude of the seismograms are very similar; however, the effect of the "numerical noise" of the seismogram associated to the coarser discretization of 1 km of the seismic source is drastically reduced for the corresponding to the 0.25 km one. This effect is clearly shown in the Fourier Amplitude spectra of the seismograms, which are shown on the

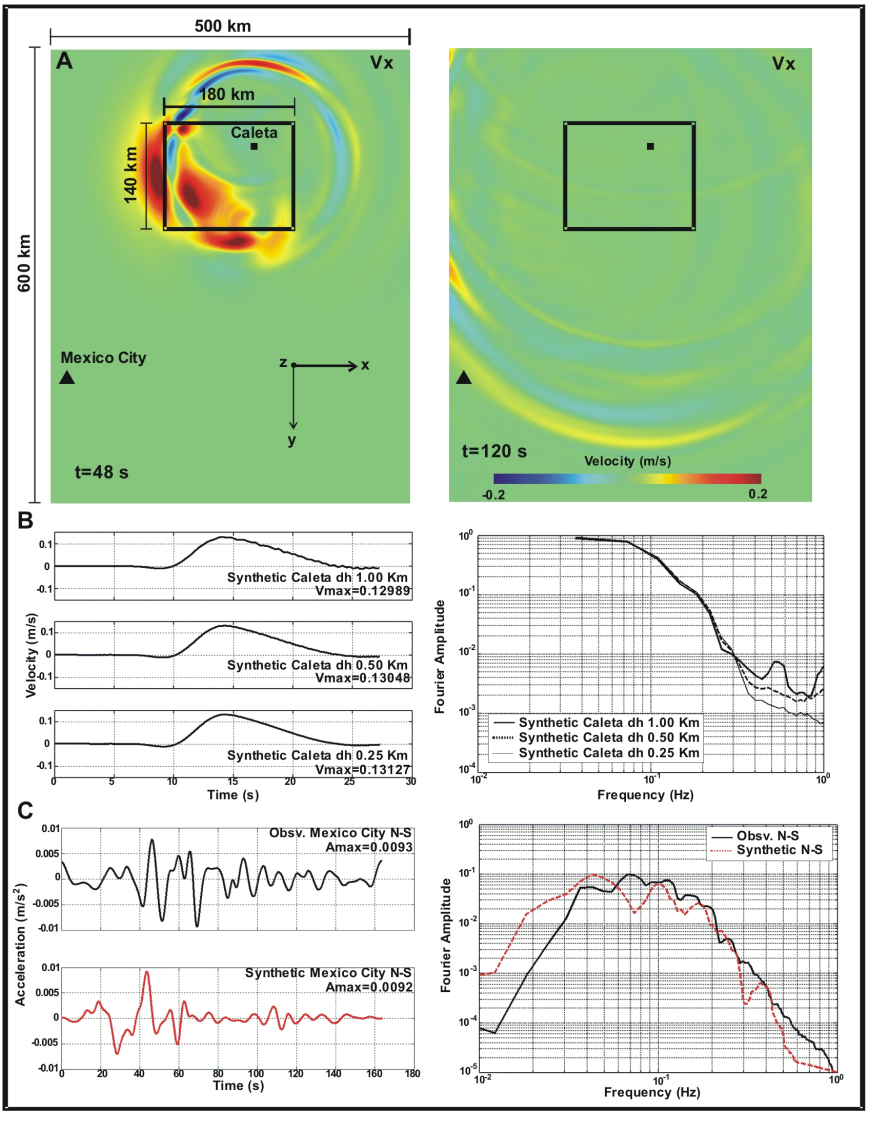


Fig. 4. A) Snapshot of the velocity wavefield in the X direction of propagation for f<= 0.2 Hz in the surface of the domain of interest; B) Left side seismogram, right side Fourier amplitude spectra obtained for the dh=1, 0.5 and 0.25 km discretizations; C) Left side observed and synthetic accelerograms north-south direction, right side Fourier amplitude spectra for the Mexico's Ms 8.1 earthquake

right side of Fig. 4B. In the latter, the mentioned "numerical noise" of the dh = 1 km discretization is associated to the bump between 0.5 and 0.6 Hertz of its Fourier amplitude spectra, versus the "no bump" at the same frequencies of the dh = 0.25 km discretization.

Finally, in Fig 4C we show the observed and synthetic (for a spatial discretization dh = 0.5km) low frequency, North-south accelerograms of the 19/09/1985 Ms 8.1 Mexico earthquake, and their corresponding Fourier Amplitude spectra for the firm soil Tacubaya site in Mexico City, i.e. at a far field observational site. Notice in these figures that the agreement between the observed and the synthetic accelerograms is reasonable both in the time and in the frequency domain.

6 Conclusions

We decomposed a realistic-size domain in 1D, 2D and 3D using data parallelism. Each processor allocates memory for its own subdomain and two plane faces of padding for each face in order to compute independently the finite difference calculation. We improve I/O using collective communications, but they are no reported in getting the performance of the implementation. The efficiency achieved is of 94% for 128 and of 85% extrapolating to 1,024 processors of the HP Cluster Platform 4000 Opteron dual core supercomputer of UNAM. The low frequency synthetic seismograms obtained with the parallel code implemented, particularly the ones for a spatial discretization of 0.5 and 0.25km show a good fit, both in the time and in the frequency domain with the observations of the Mexico's 19/09/1985 Ms 8.1 subduction earthquake.

Acknowledgments

We acknowledge DGSCA, UNAM for the support we received to use the HP Cluster Platform 4000 Opteron dual core supercomputer (KanBalam). We would also like to thank the staff of the Supercomputing Department at DGSCA, UNAM, particularly to José Luis Gordillo, Eduardo Murrieta and Adrián Durán. Thanks to Martha Mora for her grammatical suggestions.

References

[1] Chavez, M., Olsen, K., Cabrera, E.: Broadband Modeling of Strong Ground Motions for Prediction Purposes for Subduction Earthquakes Occurring in the Colima-Jalisco Region of Mexico. 13WCEE, Vancouver, B.C., Canada, August 1-6, 2004, Paper No 1653 (2004)

[2] Minkoff, S.E.: Spatial Parallelism of a 3D Finite Difference Velocity-Stress Elastic Wave Propagation code. SIAM J. Sci. Comput. 24(1), 1–19 (2002)

[3] Madariaga, R.: Dynamics of an Expanding Circular Fault. Bull. Seismol. Soc. Amer. 66, 639–666 (1976)

[4] Mendoza, C., Hartzell, S.: Slip Distribution of the 19 September 1985 Michoacan, Mexico, Earthquake: near Source and Telesismic constrains. Bull. Seismol. Soc. Amer. 79, 655–699 (1989)

[5] Amdahl, G.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: Conference Proceedings, AFIPS, pp. 483-485 (1967)

[6] Supercomputing Department, DGSCA, UNAM, Supercomputer KanBalam http://www.super.unam.mx/index.php?op=eqhw

Virtual Parallel Machines Through Virtualization: Impact on MPI Executions

Benjamin Quetier[1], Thomas Herault[1], Vincent Neri[1], and Franck Cappello[2]

[1] Univ Paris Sud; LRI; INRIA Futurs; F-91405 Orsay France
quetier@lri.fr, herault@lri.fr, neri@lri.fr
[2] INRIA Futurs; F-91405 Orsay France
fci@lri.fr

1 Introduction

Virtual Machines (VM) are used to provide homogeneous environments at low costs, enhanced security of execution through confinment of the application, and sometimes for enabling checkpointing capabilities. They rely on special hardware instructions, or pure software implementations, and are usually located between the hardware and the different operating systems. In this work, we evaluate the impact of virtualization parameter (like the number of VM per physical machine) over applications and micro-benchmarks running inside Message Passing Interface environments to determine the feasability and efficiency of virtual environments for high performance computation emulation.

We have used the Xen[2] paravirtualization tool to virtualize up to 64 VMs in a single physical one. Xen is an open source virtual machine monitor for the Linux operating system which reports low overhead and efficient execution of Linux. In this work, we study the impact of the virtualization degree (the number of VM per physical machine) and different disposition of these VMs.

2 Methodology and Hardware Platform

Our experimental platform consists of a subset of Grid'5000 [1]: GriD eXplorer. We have used 256 dual AMD opteron 246 at 2GHz with 2MBytes of memory and with a gigabit network. For the Virtualization, we have used Xen3.04. For the host and guests system are based on Debian etch with 2.6.16.33-xen kernel. The version of MPI is : MPICH-1.2.7p1.

On this platform we have used two NAS benchmark [3]: block tridiagonal solver (BT) and conjugate gradient (CG) compiled in the B class for 256 processes. We have tested various VM configurations: 4 variation of the virtualization degree and 2 slicing of BT and CG datas on these VMs.

3 Results

The figure show the execution of BT and CG in 4 virtual machines configurations: 8 host times 32 VMs, 16 hosts times 16 Vms, 32 hosts times 8 VMs and 64 hosts

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 381–383, 2007.

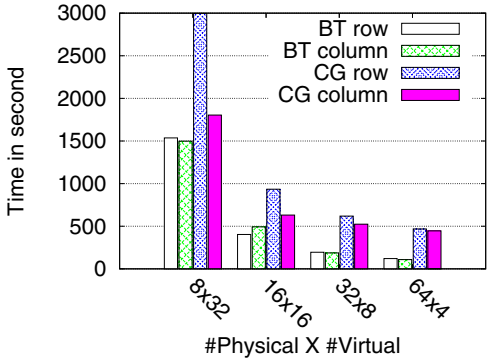


Fig. 1. Virtual machines placements and repliments

times 4 VM so each time a total of 256 VMs. We also use 2 VM placements: by row and by column.For the placement by row we number the machine list by taking the first VM of each physical machine then the second VM of each, ... For the colum one we take all the VM of the first physical machine, then all the VM off the second, ... We can notice many things:

- The execution time decreases in comparison with the number off physical machines in the experiment but not in a linear way. Indeed, the application is compiled for 256 process but for example in the case 8x32 of the figure, there are only 16 physical processors so there are some computation overhead and some network overhead due to the slicing of the problem.
- For the CG benchmark which is a very communicating application, the difference between the placement in row, and the placement in column is clearly visible. For the case 8x32 for example, we have 2996 seconds for the row placement and 1804 for the column so a 40% gain. This can be explain by the communication model of CG in which it seems there are more communications between a process and his neighborhood than between this process and farther other processes. In fact, in the column placement, process' neighbors are regulary in VMs of the same phycical machine where internal communication are faster (in term of latency and bandwidth).
- For the BT benchmark which is a low communicate application, there are nearly not difference between row and column placement.

References

1. Cappello, F., Desprez, F., Dayde, M., Jeannot, E., Jegou, Y., Lanteri, S., Melab, N., Namyst, R., Primet, P.V.B., Richard, O., Caron, E., Leduc, J., Mornet, G.: Grid5000: a nation wide experimental grid testbed. International Journal on High Performance Computing Applications (2006)

2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T.m, Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 164–177. ACM Press, New York (2003)
3. Bailey, D.H., Barton, J.T.: The nas kernel benchmark program. Technical Report TM-86711, NASA (August 1985)

Seshat Collects MPI Traces: Extended Abstract

Rolf Riesen*

Sandia National Laboratories
rolf@sandia.gov

Traces collected at the MPI level can help understand the behavior of applications by using these traces to visualize the communication patterns of an application. The traces can also be used for debugging and as input to system simulators. These trace driven simulators can help with learning how an application makes use of the communication fabric and how an application will perform on a next-generation machine.

Tools that collect MPI traces have one these two drawbacks: They do not produce accurate enough, fine grained traces, or they distort the application behavior and run time. Tools that generate detailed traces influence the run time, and sometimes the behavior, of the application under measurement, because the amount of data collected is large and requires time to send to storage [1]. The timestamps in the trace data are influenced as well. Tools that are less intrusive, collect less, or less accurate, information [2,3,4]. We propose a method to solve both of these problems.

Some users of trace data are interested in the message data itself. For example, a trace driven simulator that simulates one node of a parallel application needs to feed the process on that node valid data. Otherwise the process might not behave in the same way as it would outside the simulator, when it is running as part of a parallel application. Collecting the application data of every MPI message during an application run generates enormous trace files and greatly influences the timing of an application.

This extended abstract describes a tool named Seshat[1] [5] which we have extended to allow tracing of MPI applications. Seshat is an execution-driven network simulator with a feedback channel into the application that it uses to update the virtual time the application is running in. It is written as a library that is linked with an MPI application. No instrumentation of the application code is necessary; relinking it with Seshat is enough. Seshat makes use of the profiling interface that is part of the MPI standard (PMPI). With hooks into most of the MPI calls, Seshat is able to initialize itself, collect information about the running application, and adjust the application's virtual timeframe.

The application runs as before, but on an additional node runs the Seshat network simulator. All MPI messages are sent and received as before, but they also generate events that are sent to the simulator. The simulator calculates the time this message was supposed to spend in the simulated network and informs the receiver how much virtual time has elapsed since the message was sent. The receiver uses that information to update its local virtual clock. In effect, we can simulate a different network than the

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

[1] Seshat was the Egyptian goddess of measurement and recording.

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 384–386, 2007.

one we are running on, and use Lamport's time synchronization mechanism [6] to keep the application unaware of the wall-clock time.

This independence of the time system the application runs in, plus the knowledge about every single message of the application under test, allows the network simulator to generate MPI traces. For proof of concept we built a prototype that writes a 90-byte ASCII text line to a file for every message event that arrives at the network simulator. Writing this data slows down the simulator when measured in wall-clock time. However, the virtual time-frame the application is running in, has not changed. We have conducted several experiments to test this hypothesis.

We ran experiments on Sandia's Cray XT3™ Red Storm machine. It was running version 1.5.39 of the Cray system software. The performance characteristics of that software release are also the ones Seshat simulates. We used the NAS parallel benchmarks version 3.2.1 to verify our claims. These benchmarks are simple compared to real applications. However, we are only interested in a proof of concept. Any code that sends and receives a large number of MPI messages will do. In some tests we write so much information that we slow down the benchmark so it takes several hours of wall-clock time to complete. Yet, it reports the same few seconds or minutes of (virtual) run time as it would when run natively.

When tracing is enabled, the LU, class A benchmark reports (virtual) times that are within 6% of what it reports when run in native mode. (In most cases within 2%.) The wall-clock time, however, is 2,600 times higher than the 64-node native run. This could be lowered by making the Seshat network simulator parallel and have it write to a high-performance file system. The fact that this simple experiment, writing to an NFS-mount file system from a single simulator node, does not change the time the LU benchmark reports, shows that our approach works. It will let us collect huge traces that take a long time to write to stable storage, without changing the time-related behavior of the application under test.

Unfortunately, there is is currently a bug in Seshat that prevents it from performing as well as LU for some of the other NAS parallel benchmarks. The CG benchmark, for example, reports widely different times when attached to Seshat, then when it runs natively. We know this is a Seshat virtual time bug, and is not due to tracing. Although the virtual time reported when CG runs under Seshat is wrong, it does not change when we enable tracing. That means our mechanism for collecting large traces without influencing the application works. We are investigating the problem in Seshat's virtual time routines and will fix it.

References

1. Chung, I.H., Walkup, R.E., Wen, H.F., You, H.: MPI performance analysis tool on Blue Gene/L. In: Proc. IEEE/ACM SuperComputing, Tampa, FL (November 2006)
2. Vetter, J.S., Yoo, A.: An empirical performance evaluation of scalable scientific applications. In: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA (2002)
3. Noeth, M., Mueller, F., Schulz, M., de Supinski, B.R.: Scalable compression and replay of communication traces in massively parallel environments. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS) (2007)

4. Knüpfer, A., Nagel, W.E.: Compressible memory data structures for event-based trace analysis. Future Generation Computer System 22(3), 359–368 (2006)
5. Riesen, R.: A hybrid MPI simulator. In: IEEE International Conference on Cluster Computing (CLUSTER'06) (2006)
6. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)

Dynamic Optimization of Load Balance in MPI Broadcast

Takesi Soga[1], Kouji Kurihara[2], Takeshi Nanri[3], Motoyoshi Kurokawa[4], and Kazuaki Murakami[2]

[1] Fukuoka IST, 3-8-33 Momochihama, Sawara-ku, Fukuoka 814-0001, Japan
soga@ist.or.jp
[2] Graduate School of Information Science and Electrical Enginnering, Kyushu University
kurihara@c.csce.kyushu-u.ac.jp,
murakami@i.kyshu-u.ac.jp
[3] Computing and Communications Center, kyushu University
nanri@cc.kyushu-u.ac.jp,
[4] Advanced Center for Computing and Communication, RIKEN, 2-1 Hirosawa, Wako, Saitama, Japan
motoyosi@riken.jp

Abstract. There are many algorithms that compose broadcast from point-to-point communications, such as Binary Tree and Binomial Tree. Though many implementations of these algorithms are proposed in MPI libraries like MPICH [1], most of them are based on an assumption that all processes begin the broadcast at the same time. That means the orders of the point-to-point communications in the broadcast are arranged numerically, according to the rank of each process. However, naturally each process starts broadcast at different times, mainly because of the imbalance of workload of each process. That causes unnecessary waiting time on processes. Also, in a broadcast algorithm such as binomial tree algorithm, the amount of communication is different for each process therefore load imbalance is increased by the occurrence of both heavy workload and heavy communications on a same process. Our method purposes to solve these problems dynamically. This method solves these problems by profiling the workload of each rank at runtime and adjusting the orders of point-to-point communications according to the information.

In the various algorithms of MPI_Bcast, binomial tree is one of the most popular one. It broadcasts a message to all M processes in the group with logM steps of point-to-point communications. At each step, each process that has already received the data sends data to the process which has not received yet.

Because the number of send operations is different in each rank, if a heavy workload is assigned to the rank that invokes many send operations in the tree, whole load-imbalance causes the longer wait time at the ranks that receives the message from the heavy-loaded rank.

Also, the performance of the broadcast changes according to the starting time of broadcast at each rank, even if the same algorithm is chosen. The difference of the starting times is caused mainly by the load-imbalance such as the difference of instruction counts or cache efficiency at each node. Generally, these kinds of

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 387–388, 2007.

differences are not easy to predict before executing programs. Therefore, it is important to consider the better implementation of the algorithm according to the behavior of the program at runtime.

To adjust the implementation of algorithm to the behavior of the program, we introduce a virtual rank and a virtual rank table. The virtual rank represents the positions of processes in the collective communication. And the virtual rank table that maps real ranks to the virtual ranks. The implementation can be adjusted according to the load balance of each rank by changing the entries of the virtual rank table. The amount of the load of each rank is determined by the waiting time in MPI_Bcast. Virtual ranks that receive the message in earlier steps are responsible for larger numbers of sends to other ranks. Therefore, by mapping the real ranks with longer waiting time in previous MPI_Bcasts to the virtual ranks that receive the message earlier, the total waiting time can be reduced.

This dynamic optimization method changes the entries as follows. At first, the wait time of each rank is measured from the wait operation for the receive request in each MPI_Bcast. Then, this wait time is compared with that of previous MPI_Bcast. If the difference is larger than a threshold, the rank calls MPI_Put to send the information of the wait time to the optimizer rank. Once after N times of MPI_Bcast, the optimization is executed on the optimizer rank. From the information arrived so far, it finds the rank with minimum wait time and that with maximum wait time and mark them as a candidate to exchange the entries of the virtual rank table. After the optimization phase, the application phase is executed on all the rank in the communicator. In this time, the information of the pair of the ranks that exchange the entry of the table and the count N that shows next optimization time is propagated. On arrival of the information, each rank exchanges the entries of the virtual rank table of its own according to the information.

This method has been build experimentally on RSCC(RIKEN Super Combined Cluster) at RIKEN Japan. The experiment uses an MPI program that invokes MPI_Bcast. In addition to that, a pseudo load is executed before each MPI_Bcast on the rank of the first receiver of the message from the root rank in the original binomial tree. Therefore, extra load on this rank is critical to the entire performance of MPI_Bcast. This experiment shows that the overall execution time of the MPI_Bcast can be reduced by around 40%.

Acknowledgements

This work has been supported by the Petascale System Interconnect project on "Fundamental Technologies for the Next Generation Supercomputing" of MEXT (Ministry of Education, Culture, Sports and Technology) Japan. We thank for computational resources of the RIKEN Super Combined Cluster (RSCC) also.

Reference

1. MPICH http://www-unix.mcs.anl.gov/mpi/mpich

An Empirical Study of Optimization in Seamless Remote MPI-I/O for Long Latency Network

Yuichi Tsujita

Department of Electronic Engineering and Computer Science,
School of Engineering, Kinki University
1 Umenobe, Takaya, Higashi-Hiroshima, Hiroshima 739-2116, Japan
tsujita@hiro.kindai.ac.jp

Abstract. A Stampi library realizes MPI-I/O operations among computers which have different MPI libraries by bridging the both libraries with TCP sockets. If interconnections among computers have long latency, throughput is degraded due to unoptimized configuration of TCP sockets. For effective remote MPI-I/O operations, improvement in throughput is an important issue. In this research work, I/O performance was measured on interconnected PC clusters which were in different network segments to find desirable configuration. We notice that optimization in socket buffer sizes and precise traffic control were quite effective to achieve high throughput I/O.

Keywords: MPI-I/O, Stampi, MPI-I/O process, router process, PSPacer.

Stampi was originally developed to support seamless MPI communications among different MPI libraries. MPI communications among them are realized by deploying a wrapper interface library on top of each MPI library [1]. It also supports MPI-I/O operations both inside a computer and among computers which have different MPI libraries by invoking MPI-I/O processes which play I/O operations on a remote computer [2]. Its remote MPI-I/O operations by using MPICH [3] were realized on a PVFS2 file system [4] with a support of ROMIO [5] to use huge storage space of the file system. It provided sufficient performance in a LAN environment even if derived data types were used. If network connections have long latency like WAN, throughput is degraded due to unoptimized socket buffer size, for example. It is well known that applying twice the product of bandwidth and latency in a socket buffer size provides better performance. In addition, pacing TCP packets improves throughput by using a loadable Linux kernel module of PSPacer [6] which smooths bursting traffic without any special hardware. To improve throughput of the remote I/O operations, we have studied a desirable solution by tuning the buffer size and applying the PSPacer.

We have evaluated its performance in interconnected two Linux PC clusters. All the PC nodes were connected via Gigabit Ethernet switches. As a native MPI library, MPICH-1.2.7p1 was used. One of the clusters had a PVFS2 file system (version 1.5.1) for I/O operations by MPI-I/O processes. One FreeBSD PC server

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 389–390, 2007.

was located between the clusters as a gateway with 1 Gbps connections. Several network latencies were applied to the network connections among the clusters by dummynet [7] of the PC server.

In performance measurement of remote I/O operations, user processes were initiated by a Stampi's start-up command on one of the clusters and the same number of MPI-I/O processes were invoked on the other by rsh. Firstly, visible I/O times of collective I/O functions for 5 ms latency were measured in terms of socket buffer sizes with and without the PSPacer. In this case, applying around twice the product of bandwidth and latency minimized the times, but there was not significant difference in the times with and without the PSPacer support. Secondly, the same measurement was carried out for 50 ms latency. The same approach in tuning the socket buffer size minimized the times. In addition, the times were minimized by using the PSPacer relative to the times without it in write operations. For example, the time with it were around 30 % of that without it for 256 Mbyte message data when 12 Mbyte was applied to a socket buffer size.

At the present, we can not control configuration of PSPacer in a user program. As a future work, implementation of a tuning mechanism for it in the remote I/O system is considered by using a libnl library [8] of it.

Finally the author would like to thank the staff at Center for Computational Science and e-Systems (CCSE), Japan Atomic Energy Agency (JAEA), for providing a Stampi library and giving useful information. This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Grant-in-Aid for Young Scientists (B), 18700074.

References

1. Imamura, T., Tsujita, Y., Koide, H., Takemiya, H.: An architecture of Stampi: MPI library on a cluster of parallel computers. In: Dongarra, J.J., Kacsuk, P., Podhorszki, N. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1908, pp. 200–207. Springer, Heidelberg (2000)
2. Tsujita, Y., Imamura, T., Takemiya, H., Yamagishi, N.: Stampi-I/O: A flexible parallel-I/O library for heterogeneous computing environment. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J.J., Volkert, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2474, pp. 288–295. Springer, Heidelberg (2002)
3. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI Message-Passing Interface standard. Parallel Computing 22, 789–828 (1996)
4. PVFS2: http://www.pvfs.org/pvfs2/
5. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems, pp. 23–32 (1999)
6. Takano, R., Kudoh, T., Kodama, Y., Matsuda, M., Okazaki, F., Ishikawa, Y.: Improving TCP performance by using precise software pacing method. IEICE Technical Report 105, 29–32 (2006) (in Japanese)
7. dummynet: http://info.iet.unipi.it/~luigi/ip_dummynet/
8. libnl - netlink library: http://people.suug.ch/~tgr/libnl/

Multithreaded and Distributed Simulation of Large Biological Neuronal Networks

Jochen M. Eppler[1,4], Hans E. Plesser[2], Abigail Morrison[3], Markus Diesmann[3,4], and Marc-Oliver Gewaltig[1,4]

[1] Honda Research Institute, Offenbach/Main, Germany
eppler@biologie.uni-freiburg.de
[2] Dept. of Mathematical Sciences and Technology, Norwegian University of Life Sciences, PO Box 5003, 1432 Ås, Norway
[3] Computational Neuroscience Group, RIKEN Brain Science Institute, Wako-shi, Saitama, Japan
[4] Bernstein Center for Computational Neuroscience, Albert-Ludwigs-University, Freiburg, Germany

1 Introduction

To understand the principles of information processing in the brain, we depend on models with more than 10^5 neurons and 10^9 connections [1]. These networks can be described as graphs of threshold elements that exchange point events.

From the computer science perspective, the key challenges are to represent the connections succinctly and to transmit events and update neuron states efficiently. We present the Neural Simulation Tool NEST (www.nest-initiative.org, [2]), a neuronal network simulator which addresses all these requirements. To simulate very large networks in acceptable time and with acceptable memory requirements, NEST uses a hybrid strategy, combining distributed simulation across cluster nodes (MPI) with thread-based simulation on each computer.

2 Network Representation and Update

Conceptually, NEST represents the network as a list of nodes. Nodes are either neuron models, devices for recording and stimulation, or sub-networks and are assigned to one of N_{VP} virtual processes, using a simple modulo algorithm [3]. A virtual process (VP) is a POSIX thread that lives in one of N_{MPI} MPI processes. Each of the processes contains the same number of threads, N_{Thrd}. Device nodes are created for each virtual process to allow parallel data i/o. This is particularly important for device nodes that have to deliver large amounts of data to their targets. To balance the load of all virtual processes, neurons are only created on the virtual process they are assigned to. On all other virtual processes, they have light-weight proxies. Each node or proxy only stores the subset of connections that reach nodes (but not proxies) on the same virtual process. Thus, the network connections are also distributed, while cache problems are reduced to a minimum.

NEST evaluates the network model on an evenly spaced time-grid $t_i := i \cdot \Delta$, where Δ is determined by the shortest transmission delay in the system. At each

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 391–392, 2007.

point, the network is in a well-defined state S_i. Starting at an initial state S_0, a global state transfer function $\mathrm{U}(S)$ propagates the system from one state to the next, such that $S_{t+\Delta} \leftarrow \mathrm{U}(S_t)$. As a side effect of $\mathrm{U}(S_t)$, nodes create events that must be delivered to the target nodes after a delay that depends on the connection. The network model in NEST is evaluated by executing the following algorithm:

1: $t \leftarrow 0$
2: **while** $t < T_{\mathrm{stop}}$ **do**
3: **parallel on all** VP **do**
4: deliver all events due
5: call $\mathrm{U}(S_t)$ for all nodes
6: **end parallel**
7: exchange events between VPs
8: increment network time: $t \leftarrow t + \Delta$
9: **end while**

The optimized data structures used for communication are described in [3].

3 Results

We demonstrate the performance of NEST, using a benchmark simulation of a large biological neural network model. We show that NEST scales supra-linearly for different combinations of threads and MPI processes.

On a cluster with 96 processor cores in 24 compute nodes and a central Infiniband switch we achieve real time with a network of 10^5 neurons with 10^9 synapses. On this architecture, the `MPI_Allgather` function performs better than the CPEX algorithm [4]. We are now investigating how different implementations of Allgather influence the performance of our multi-threaded/distributed communication scheme.

Acknowledgments. This work was partially funded by DAAD/NFR 313-PPP-N4-lk, DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, and EU Grant 15879 (FACETS).

References

1. Abeles, M.: Corticonics: Neural Circuits of the Cerebral Cortex, 1st edn. Cambridge University Press, Cambridge (1991)
2. Gewaltig, M.O., Diesmann, M.: Scholarpedia (2007), http://www.scholarpedia.org/article/NEST_(Neural_Simulation_Tool)
3. Morrison, A., Mehring, C., Geisel, T., Aertsen, A., Diesmann, M.: Advancing the boundaries of high connectivity network simulation with distributed computing. Neural Computation 17(8), 1776–1801 (2005)
4. Tam, A., Wang, C.: Efficient scheduling of complete exchange on clusters. In: 13th International Conference on Parallel and Distributed Computing Systems (PDCS 2000), Las Vegas (August (2000)

Grid Services for MPI

Camille Coti[2], Ala Rezmerita[1], Thomas Herault[1], and Franck Cappello[2]

[1] Univ Paris Sud; LRI; INRIA Futurs; F-91405 Orsay France
[2] INRIA Futurs; F-91405 Orsay France
coti@lri.fr, rezmerit@lri.fr, herault@lri.fr, fci@lri.fr

1 Introduction

Institutional grids consist of the aggregation of clusters belonging to different administrative domains that build a single parallel machine. In order to run a MPI application over an institutional grid, one has to address many problems. One of the first problems to solve is the connectivity of nodes not belonging to the same administrative domain.

To protect the network from unauthorized access many sites use firewalls. On some sites firewalls are configured to allow outbound connections and to block inbound connections, often with the exception of a few well-known ports (*e.g.*, SSH). On some other sites there is a strict separation between the internal and external networks and only a front-end machine is accessible. Theses connectivity constraints limit the execution of parallel application between multiple sites.

The connectivity problems can sometimes be solved when only one site uses a firewall, since all required connections are initiated from the protected site. However this solution requires modifications to applications or communication libraries. Also, if all sites are using firewalls this approach can no longer be applied. Another solution is to configure the firewalls so that a port range is open and adapt the applications to use only theses ports. However this solution is a threat to the site security. Sometimes the only possibility for the compute nodes to communicate with the outside world is to use the front-end machine as a bridge. In addition to causing connection setup problems the use of Network Address Translation (NAT) devices complicates machine identification. The private addresses used in a NAT site are not globally unique, which causes difficulties in creating a unique identifier for every machine.

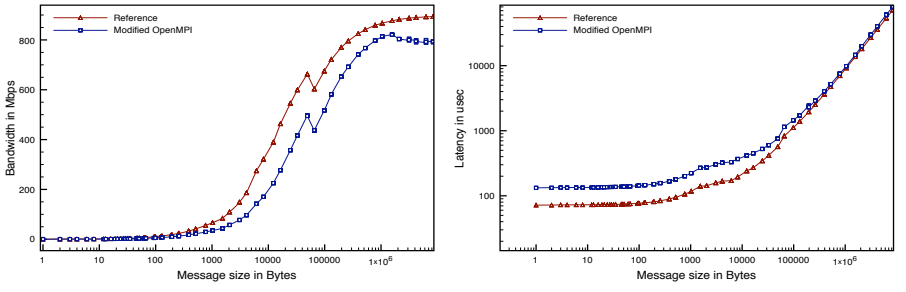
In this work, we propose a set of Grid or Web Services that provide a new level of communication for establishing connectivity of MPI applications over an experimental grid.

2 Results

We define a distributed framework to allow the grid infrastructure to provide services to the applications. In this paper we detail a brokering service that provides the computing nodes a way to communicate with each other. Other services can be implemented in our framework, such as monitoring service, spawning service and distributed storage service. The brokering service establishes a connection between nodes that would not be able to communicate with each other otherwise because a NAT and/or a firewall

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 393–394, 2007.

are standing between them. When the MPI library needs to establish a communication between two nodes that don't belong to the same cluster, it invokes the brokering service that finds the best method to establish this connection (NAT and/or firewall bypassing) and returns the appropriate connection information to the initiator of the connection. Some techniques have been presented in [2]. We implemented the service using the light-weight web-services engine gSOAP[3] and interfaced it with OpenMPI[1].



(a) Bandwidth measured by NetPIPE (b) Latency measured by NetPIPE

Fig. 1. Communication performance

Figure 1 shows the impact of the framework on communication performances, measured using the NetPipe test. The nodes are interconnected by a proxy, which adds a hop between them. We can see the impact of this additional hop on bandwidth on figure 1(a) and on latency on figure 1(b). Since the service is invoked only to establish the communication, it has no effect on the performances of the communications themselves. Therefore, the other techniques that establish a direct connection between two nodes (reverse connection, traversing TCP and TCP hole punching) give the same performances as a direct connection without a firewall. The additional cost induced by the establishment of the connection is 10 ms.

References

1. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104 (September 2004)
2. Rezmerita, A., Morlier, T., Néri, V., Cappello, F.: Private virtual cluster: Infrastructure and protocol for instant grids. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 393–404. Springer, Heidelberg (2006)
3. van Engelen, R.: Pushing the SOAP envelope with web services for scientific computing. In: proceedings of the International Conference on Web Services (ICWS), pp. 346–352 (2003)

Author Index