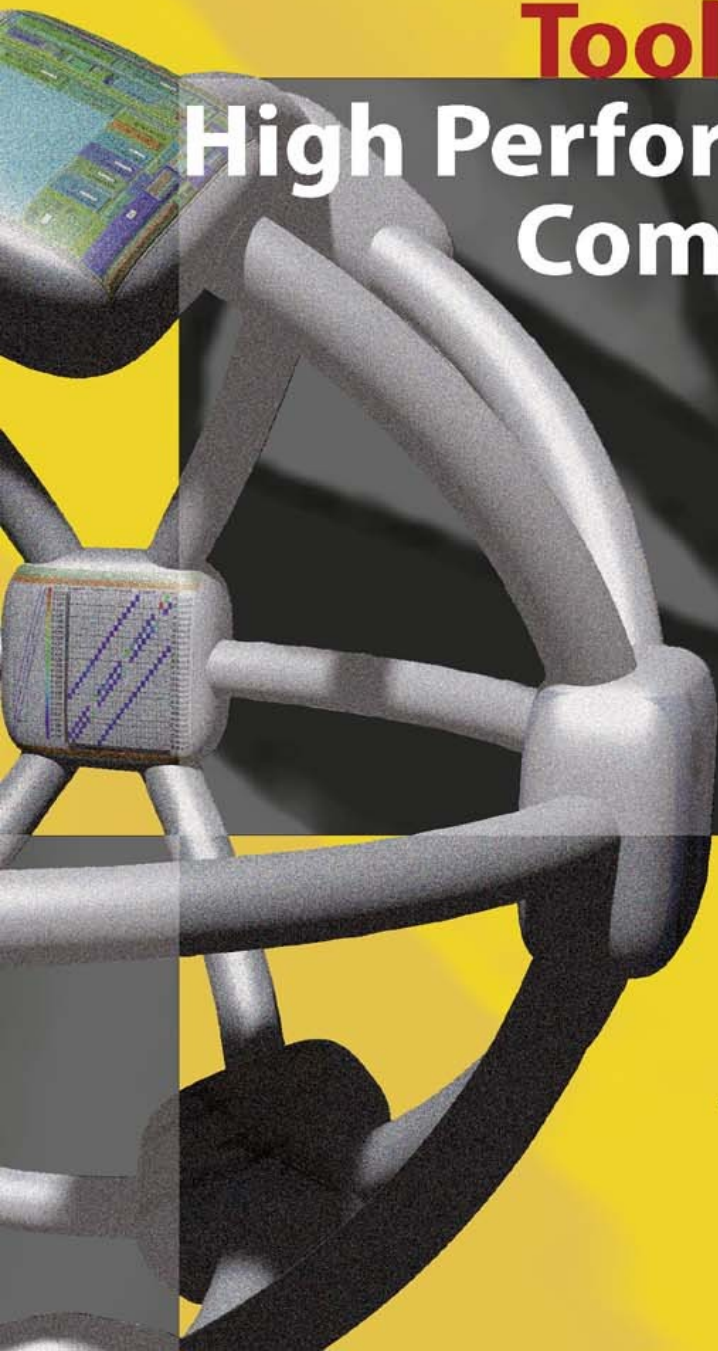
Michael Resch · Rainer Keller
Valentin Himmler · Bettina Krammer
Alexander Schulz *Editors*

Tools for High Performance Computing



H L R S

Springer

Tools for High Performance Computing

Michael Resch · Rainer Keller · Valentin Himmler ·
Bettina Krammer · Alexander Schulz
Editors

Tools for High Performance Computing

Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart

Springer

Michael Resch, resch@hlrs.de
Rainer Keller, keller@hlrs.de
Valentin Himmler, himmler@hlrs.de
Bettina Krammer, krammer@hlrs.de
Alexander Schulz, schulz@hlrs.de

Höchstleistungsrechenzentrum
Stuttgart (HLRS)
Nobelstr. 19
70569 Stuttgart
Germany

Front cover figure: Visualisation of Parallel Tools for HPC, Vampir, Totalview, Acumem, Kcachegrind and the NEC SX-8

ISBN 978-3-540-68561-6 e-ISBN 978-3-540-68564-7

DOI 10.1007/978-3-540-68564-7

Library of Congress Control Number: 2008927892

Mathematics Subject Classification (2000): 68-06, 68N18, 68Q85 68Q60, 68U99, 94A99

Cover design: WMXDesign, Heidelberg

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Preface

Developing software for current and especially for future architectures will require knowledge about parallel programming techniques of applications and library programmers. Multi-core processors are already available today, and processors with a dozen and more cores are on the horizon.

The major driving force in hardware development, the game industry, has already shown interest in using parallel programming paradigms, such as OpenMP for further developments. Therefore developers have to be supported in the even more complex task of programming for these new architectures.

HLRS has a long-lasting tradition of providing its user community with the most up-to-date software tools. Additionally, important research and development projects are worked on at the center: among the software packages developed are the MPI correctness checker Marmot, the OpenMP validation suite and the MPI-implementations PACX-MPI and Open MPI. All of these software packages are being extended in the context of German and European community research projects, such as ParMA, the InterActive European Grid (I2G) project and the German Collaborative Research Center (Sonderforschungsbereich 716). Furthermore, industrial collaborations, i.e. with Intel and Microsoft allow HLRS to get its software production-grade ready.

In April 2007, a European project on Parallel Programming for Multi-core Architectures, in short ParMA was launched, with a major focus on providing and developing tools for parallel programming.

This project is funded through the ITEA initiative and involves partners from industry and research from application providers and tools developers, such as platform provider Bull, Allinea with its parallel debugger DDT, the Center for Information Services and High Performance Computing (ZIH) with the parallel performance analyser Vampir-NG and the Central Institute for Applied Mathematics (ZAM) with Kojak/Scalasca.

As a spin-off of all these activities the 1^{st} Parallel Tools Workshop was held on 7-9th of July, 2007 at the High-Performance Computing Center Stuttgart (HLRS). Participants from research and developers from science and industry were invited to this interactive workshop which attracted 67 scientists from all over the world.

The focus was on presentations on the various tools, but also on giving hands-on sessions to demonstrate the strengths of each tool.

With this year's 2^{nd} Parallel Tools Workshop on July the 7th/8th, HLRS wants to offer its industrial and scientific user community, precisely this information in the form of a thorough publication on the software packages, again ranging from debugging tools to performance analysis and best practices in integrated developing environments for parallel platforms. The papers of this workshop are presented here. Last year's workshop brought together software developers from the US, Germany, France and Great Britain, and we expect an even wider audience this year.

This year's contributions are in the fields of Integrated Development Environments, Parallel Debugging and Performance Analysis tools from a wide range of scientific and industrial tool developers. This includes tools from vendors such as Cray, Intel, IBM, Sun, Acumem, Allinea and Totalview, as well as research institutions, including the University of Oregon, Technical University of Dresden and the Research Center in Juelich.

Stuttgart, April 2008

Michael Resch, *Rainer Keller*
Valentin Himmler, *Bettina Krammer*
Alexander Schulz

Contents

III Performance Analysis Tools

Parallel Performance Analysis Tools

List of Contributors

I
Integrated Development Environments

Sun HPC ClusterTools™ 7+: A Binary Distribution of Open MPI

Terry Dontje, Don Kerr, Daniel Lacher, Pak Lui, Ethan Mallove, Karen Norteman, Rolf Vandevaart, and Leonard Wisniewski

Abstract The Sun HPC ClusterTools 7 release was Sun's first binary distribution of the Open MPI software. This release marked a change in source-code base for Sun from a proprietary code base derived from the Thinking Machines Corporation Globalworks™ software to the open-source Open MPI software. Sun HPC ClusterTools includes packages of binaries built from the Open MPI source code by the Sun™ Studio compilers and install scripts for installing those packages across a cluster of nodes. The Sun HPC ClusterTools team contributed a Sun Grid Engine plug-in and developed the uDAPL Byte Transfer Layer module as its Infiniband solution on Solaris™ operating system. Additionally, Sun HPC ClusterTools includes examples of using DTrace to analyze performance and debug MPI applications. Other product-focused activity included significant contribution to the development of the MPI Test Tool (MTT) and development of a set of user documentation. This paper describes the new Sun HPC ClusterTools based on Open MPI, focusing on areas where Sun has contributed to Open MPI.

1 Introduction

In April 2006, Sun joined the Open MPI community and decided to use Open MPI as the source-code base for its Sun HPC ClusterTools product, thereby replacing its previous proprietary source-code base [1].

Open MPI was founded by researchers at Indiana University, University of Tennessee, Los Alamos National Laboratory, and HLRS / University of Stuttgart. The initial implementation was intended to be a clean-slate approach combining attributes of each of those institutions' previous MPI implementations [2, 3]. Today, the Open MPI community includes 15 member institutions, another 9 contributor in-

Sun Microsystems, USA, e-mail: {Terry.Dontje,Don.Kerr,Daniel.Lacher,Pak.Lui,Ethan.Mallove,Karen.Norteman,Rolf.Vandevaart,Leonard.Wisniewski}@sun.com

stitutions, as well as many individual developers and users. More information about Open MPI can be found at http://www.open-mpi.org.

Sun HPC ClusterTools includes packages of binaries built from the Open MPI source code by the Sun Studio compilers and install scripts for installing those packages across a cluster of nodes [4]. The Sun HPC ClusterTools team contributed a Sun Grid Engine plug-in and developed the uDAPL Byte Transfer Layer (BTL) module as its Infiniband solution on Solaris [5, 6]. Additionally, Sun HPC ClusterTools includes examples of using DTrace to analyze performance and debug MPI applications [7]. Other product-focused activity included significant contribution to the development of the MPI Test Tool (MTT) and development of a set of user documentation [8]. This paper describes the new Sun HPC ClusterTools based on Open MPI, focusing on areas where Sun has contributed to Open MPI.

The rest of this paper is organized as follows. Section 2 gives a brief history of the Sun HPC ClusterTools product from its proprietary era through its current Open MPI participation. Sections 3 discusses several features developed by the Sun team to augment support to include Sun systems and to utilize unique features of Sun systems. Section 4 describes some of the activities in which the team participated for the Sun HPC ClusterTools product and when applicable contributed to the community. Section 5 offers some pros and cons of using Open MPI as the source code base vs. the proprietary ClusterTools code base and Section 6 wraps up with future focus areas and conclusions.

2 History

In 1996, Sun acquired the Thinking Machines Corporation Globalworks product and team. Globalworks was a set of parallel programming tools originally intended to support a variety of operating system platforms. With the acquisition by Sun, Globalworks was renamed Sun HPC ClusterTools and targeted solely as a binary distribution of parallel programming tools for the Solaris operating system and the SPARC® systems.

The first Sun HPC ClusterTools release not only included MPI libraries and a run-time environment for launching parallel jobs as it does today, but it also included a High Performance Fortran compiler, a parallel file system, the Prism parallel debugger, the Sun Scalable Scientific Subroutine Library (S3L), and various other libraries and utilities for parallel programming and cluster management [9]. Sun delivered six releases derived from the Globalworks code base, concluding with the release of Sun HPC ClusterTools 6 in March 2006, which was the first release to support Solaris on x64. Over the course of its ten-year history, except for the MPI libraries and run-time environment, the technology for the other tools were gradually integrated into other products or disbanded.

In April 2006, Sun joined the Open MPI open-source community, as one of the first vendors to embrace Open MPI as a clean-slate modular architecture with an open community-based approach to development. In April 2007, Sun debuted the

use of the Open MPI source code as its basis in Sun HPC ClusterTools 7, a binary distribution of the Open MPI libraries and the Open Run-Time Environment (ORTE) for launching parallel jobs, based specifically on Open MPI 1.2.

Sun released a Sun HPC ClusterTools 7.1 update release in November 2007 based on Open MPI 1.2.4. Sun HPC ClusterTools 7 and 7.1 supported only Solaris 10. However, the upcoming Sun HPC ClusterTools 8 release, based on Open MPI 1.3, will be the first Sun release to support a binary distribution of Open MPI on the Linux platform and also support OpenSolaris™ [10].

3 Sun-Driven features

3.1 uDAPL Byte Transfer Layer

When Sun joined Open MPI, Solaris did not support its own implementation of the Infiniband (IB) Verbs API in contrast to the Open Fabrics Alliance (OFED) Verbs implementation commonly used on Linux and other platforms [11]. Rather, Solaris did support an implementation of the User-Level Direct Access Programming Library (uDAPL). The uDAPL API is a user-level library defined by the DAT Collaborative to provide a transport-neutral infrastructure that provides RDMA capabilities in user space [12].

In Open MPI, at the lowest-level in its MPI communication stack are protocol-specific Byte Transfer Layers (BTLs). When launching a parallel job, an Open MPI user can specify which BTLs to use for MPI communication. The uDAPL BTL was originally developed by Indiana University for Linux and adapted by Sun to support Solaris as well. To select the uDAPL BTL, a user launches a parallel job as follows:

```
mpirun -btl self,sm,udapl
```

In the scope of a community-based approach, supporting a low-level software module different than the rest of the community presents a number of challenges, with the greatest difficulty being the inability to directly leverage technological advances by other community members. To this end, in the future, Solaris will support the OFED Verbs implementation, and the ClusterTools team will consequently be able to collaborate directly with the majority of the Open MPI community on development of Infiniband support.

3.2 Sun Grid Engine plug-in

Sun Grid Engine (SGE) is a resource manager which allows tight integration with the parallel job launchers of various MPI implementations. SGE and its accompanying open-source version Open Grid Engine are widely popular as free and open

resource managers. For Open MPI, Sun added an SGE plug-in module to support the launch of Open MPI jobs on SGE.

A user can invoke an Open MPI job using SGE in several ways. The most common way to start a parallel job over SGE is by submitting a batch job. The Open MPI `mpirun` command is embedded inside a batch script that will be executed by `qsub`. This allows SGE to schedule the parallel job when there are sufficient resources available for starting the parallel job on the number of nodes requested. Using this method should give consistent and reproducible runs as all the information can be specified inside the batch script.

```
# Submit a batch job with the 'mpirun' command
# embedded in a script
shell$ cat script.sh
#!/sbin/sh
#$ -N jobname
#$ -j y
#$ -o out.$JOB_NAME.o$JOB_ID
#$ -pe orte 4
/path/to/mpirun -np $NSLOTS mpijob

shell$ qsub script.sh
```

The other way is to start an SGE interactive shell that would allow the user to log on to the head node which is responsible for starting the parallel job via `q`.

```
# Allocate an SGE interactive job with 4 slots
shell$ qsh -pe orte 4

# Now run a 4-process Open MPI job
shell$ mpirun -np $NSLOTS mpijob

# Submit an SGE and OMPI job and mpirun in one line
shell$ qrsh -V -pe orte 4 mpirun -np 4 mpijob
```

It is advantageous to run large jobs with SGE as the resource management system. SGE allows the user to have exclusive use of a set of nodes dedicated to run their code without being interfered by other users. With SGE, the user would not need to come up with an explicit list of nodes to run. This simplifies the need to parse a node list, especially on a large cluster environment on which the list of nodes could be long. Also, nodes can become unavailable, but SGE always gives you an up-to-date list of usable nodes and removes the unavailable ones from the node list.

The ability to clean up temporary space and collect standard and error outputs after each run are also particularly useful for running jobs across a large number of nodes.

There is also an ability to limit an MPI job on a subset of nodes by specifying the `mpirun` command in conjunction with a host list which identifies the subset of nodes.

SGE supports many popular operating systems. This helps to select and run code on a cluster which is comprised of heterogeneous platforms. In the current SGE 6.1, the job launching mechanism for sending parallel tasks to the execution hosts relies on the "`qrsh -inherit`" command, which is an `rsh`-based mechanism. Since the ports opened by `rsh` for each connection is limited to only 1024, for large parallel jobs that need to run across hundreds or thousands of nodes at once, an SGE cluster should be configured to use `ssh` as its backend mechanism for remote launching.

SGE gathers the resource usage from its job by appending an additional group ID to a user ID while the job is running. Hence, SGE ships with its version of the RSH daemon which includes this modification. SGE can be configured to use a vanilla version of `ssh` that does not contain any changes for proper job accounting [13]. However, to achieve job accounting with `ssh`, the code for the `ssh` daemon needs to be modified and built together with SGE. This is sometimes known as the `ssh` Tight Integration. Modifying the SGE code for the `ssh` Tight Integration used to be an audacious task with earlier SGE versions, but SGE 6.1 includes these changes to simplify the task.

3.3 Sun Studio Compiler Support

Sun offers its own suite of compilers in its Sun Studio product. Sun Studio includes C, C++, and Fortran compilers as well as the `dbx` debugger, performance libraries, a program analyzer tool, support for OpenMP programs, and an integrated development environment (IDE). Sun Studio supports both the Solaris and Linux operating systems.

Open MPI supports compilation of its source code and Open MPI applications by a number of compilers. Sun HPC ClusterTools is built using Sun Studio and supports MPI applications built with Sun Studio and linked with the ClusterTools libraries. For each release version of Sun HPC ClusterTools, consult the release notes to find out which versions of Sun Studio are supported for compiling user applications.

There are numerous challenges when adding a new compiler to the support matrix of an open-source code base. In the case of supporting Sun Studio, these challenges included adapting the code base to the stricter memory alignment required by chips, incompatibilities in support levels among different compilers for various C++ libraries and Fortran functionality, and ensuring that the best set of flags are used. All of these challenges require constant monitoring as it is not expected that community members are aware of restrictions and incompatibilities among all the compilers supported by Open MPI. To make it easiest for users to use the Sun Studio compilers, the Open MPI compiler wrappers were updated to insert the best set of flags automatically for the user at compile time. The following are examples of compiler wrapper use for Sun Studio and their corresponding translations into actual compile command lines.

```
% /opt/SUNWhpc/HPC7.0/bin/mpicc -o tmp tmp.c -showme
cc -I/opt/SUNWhpc/HPC7.0/include/openmpi
```

```
    -I/opt/SUNWhpc/HPC7.0/include -o tmp tmp.c
    -R/opt/mx/lib -R/opt/SUNWhpc/HPC7.0/lib
    -R/opt/mx/lib/sparcv9
    -R/opt/SUNWhpc/HPC7.0/lib/sparcv9
    -L/opt/SUNWhpc/HPC7.0/lib
    -lmpi -lopen-rte -lopen-pal -lsocket -lnsl -lrt -lm -ldl

% /opt/SUNWhpc/HPC7.0/bin/mpicc -o tmp -xarch=v9 tmp.c -showme
cc -I/opt/SUNWhpc/HPC7.0/include/openmpi
   -I/opt/SUNWhpc/HPC7.0/include/v9
   -o tmp -xarch=v9 tmp.c
   -R/opt/mx/lib -R/opt/SUNWhpc/HPC7.0/lib
   -R/opt/mx/lib/sparcv9
   -R/opt/SUNWhpc/HPC7.0/lib/sparcv9
   -L/opt/SUNWhpc/HPC7.0/lib/sparcv9
   -lmpi -lopen-rte -lopen-pal -lsocket -lnsl -lrt -lm -ldl
```

3.4 MPI Profiling

Starting in Sun HPC ClusterTools 8, profiling support will take greater prominence in the product. Moreover, there will be four new ways to access greater levels of profiling information:

1. via DTrace probes,
2. via PERUSE probes,
3. via VampirTrace probes, and
4. via special hooks using Sun Studio Analyzer.

DTrace is a comprehensive dynamic tracing facility debuted in Solaris 10 that can be used to examine the behavior of both user programs and the operating system itself. Sun HPC ClusterTools 7 includes some examples of using Dtrace to examine MPI programs. Sun HPC ClusterTools 8 will include some new DTrace providers to examine various MPI state at key locations in the MPI message path.

DTrace allows one to place probes into code such that a DTrace script may be used to get a view of what is happening in the code. In the case of MPI, we've chosen to piggyback on the `mpi_peruse` framework, which specifies various events within an MPI library and data available for that event. To use the `mpi_peruse` framework, one usually has to have their user code write/register callbacks to be called when a PERUSE event happens. What the DTrace `mpi_peruse` provider does is expose these events via probes thus not requiring one to write actual code to handle registration or logging info. The `mpi_peruse` provider provides probes to all of the PERUSE events that are defined in the PERUSE specification. This provider allows one to capture events such as

`PERUSE_COMM_REQ_INSERT_IN_POSTED_Q`
and
`PERUSE_COMM_REQ_REMOVE_FROM_POSTED_Q`

to keep track of the posted queue depth. Likewise one can do similar queue investigations with the events that track unexpected messages. One can also track send and receive requests.

All of this allows one to determine what may be happening within the MPI library and do it in an unobtrusive way. That is while an MPI code is running you can attach `dtrace` with a script that uses the `mpi_peruse` provider to one of the processes and garner information that you are interested in. If it turns out that you need to adjust the script you can and then reattach `dtrace` to the MPI process without disrupting the MPI job. The following example is a `netstat`-like script that shows you queue changes (requests, posted, unexpected) and data transferred by the process `dtrace` is attached to.

```
/*
 * Copyright (c) 2007-2008 Sun Microsystems, Inc.
 *                         All rights reserved.
 *                         Use is subject to license terms.
 * $COPYRIGHT$
 *
 * Additional copyrights may follow
 *
 * $HEADER$
 */

BEGIN
{
  recvs_bytes=0;
  recvs_act=0;
  recvs_posted_size=0;
  recvs_unexp_size=0;
  recvs_posted_matches=0;
  recvs_unexp_matches=0;

  sends_act=0;
  sends_bytes=0;
  output_cnt = 0;
  printf("IN(Total) Q-sizes      Q-Matches    OUT\n");
  printf("bytes act posted unexp posted unexp bytes act\n");
  printf("%5d %6d %6d %5d %6d %5d      %5d %6d \n",
          recvs_bytes, recvs_act, recvs_posted_size,
          recvs_unexp_size,
          recvs_posted_matches, recvs_unexp_matches,
          sends_bytes, sends_act);
}
/* Print Statistics every 1 sec */
profile:::tick-1sec
{
  printf("%5d %6d %6d %5d %6d %5d      %5d %6d \n",
          recvs_bytes, recvs_act, recvs_posted_size,
          recvs_unexp_size,
          recvs_posted_matches, recvs_unexp_matches,
          sends_bytes, sends_act);
  ++output_cnt;
```

```
}
profile:::tick-1sec
/output_cnt==22/
{
  printf("IN(Total) Q-sizes      Q-Matches    OUT\n");
  printf("bytes act posted unexp posted unexp bytes act\n");
  printf("%5d %6d %6d %5d %6d %5d      %5d %6d \n",
          recvs_bytes, recvs_act, recvs_posted_size,
          recvs_unexp_size,
          recvs_posted_matches, recvs_unexp_matches,
          sends_bytes, sends_act);
  output_cnt=0;
}


/* Collect Send statistics */
/* Collect Active Send Requests */
mpi__peruse$target:::PERUSE_COMM_REQ_ACTIVATE
/args[3]->mcs_op=="send"/
{
  ++sends_act;
}

/* Collect Removal of Send Requests */
mpi__peruse$target:::PERUSE_COMM_REQ_NOTIFY
/args[3]->mcs_op=="send"/
{
  --sends_act;
}

/* Collect bytes Sent */
mpi__peruse$target:::PERUSE_COMM_REQ_XFER_END
/args[3]->mcs_op=="send"/
{
  sends_bytes += args[3]->mcs_count;
}

/* Collect Active Recv Request */
mpi__peruse$target:::PERUSE_COMM_REQ_ACTIVATE
/args[3]->mcs_op=="recv"/
{
  ++recvs_act;
}

/* Collect Removal of Recv Request */
mpi__peruse$target:::PERUSE_COMM_REQ_NOTIFY
/args[3]->mcs_op=="recv"&&recvs_act>0/
{
  --recvs_act;
}

/* Collect Request Placed on Posted Q */
mpi__peruse$target:::PERUSE_COMM_REQ_INSERT_IN_POSTED_Q
/args[3]->mcs_op=="recv"/
```

```
{
  ++recvs_posted_size;
}

/* Collect Msg matched Posted Q */
mpi__peruse$target:::PERUSE_COMM_MSG_MATCH_POSTED_REQ
/args[3]->mcs_op=="recv"/
{
  ++recvs_posted_matches;
}
mpi__peruse$target:::PERUSE_COMM_MSG_MATCH_POSTED_REQ
/args[3]->mcs_op=="recv"&&recvs_posted_size>0/
{
  --recvs_posted_size;
}

/* Collect messages in unexp Q */
mpi__peruse$target:::PERUSE_COMM_MSG_INSERT_IN_UNEX_Q
/args[3]->mcs_op=="recv"/
{
  ++recvs_unexp_size;
}

/* Collect messages removed from unexp Q */
mpi__peruse$target:::PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q
/args[3]->mcs_op=="recv"&&recvs_unexp_size>0/
{
  --recvs_unexp_size;
}

/* Collect messages removed from unexp Q */
mpi__peruse$target:::PERUSE_COMM_REQ_MATCH_UNEX
/args[3]->mcs_op=="recv"/
{
  ++recvs_unexp_matches;
}

/* Collect bytes being recieved */
mpi__peruse$target:::PERUSE_COMM_REQ_XFER_CONTINUE
/args[3]->mcs_op=="recv"/
{
  recvs_bytes += args[3]->mcs_count;
}

END
{
}
```

To invoke the above DTrace script, you execute the following command.

```
% dtrace -q -p 10625 -s mpistat.d
```

```
IN(Total) Q-sizes      Q-Matches    OUT
bytes act posted unexp posted unexp bytes act
   0    0     0      0      0      0     0    0
   0    0     0      0      0      0     0    0
   0    0     0      0      0      0     0    0
   0    0     0      0      0      0     0    0
   0    0     0      0      0      0     0    0
   5    1     1      0      1      0     0    0
   5    1     1      0      1      0     0    0
   5    1     1      0      1      0     0    0
   5    1     1      0      1      0     0    0
   5    1     1      0      1      0     0    0
   5    1     1      0      1      0     0    0
   5    1     1      0      1      0     0    0
   5    1     1      0      1      0     0    0
  10    1     1      0      2      0     0    0
  10    1     1      0      2      0     0    0
  10    1     1      0      2      0     0    0
  10    1     1      0      2      0     0    0
  10    1     1      0      2      0     0    0
  10    1     1      0      2      0     0    0
  10    1     1      0      2      0     0    0
  10    1     1      0      2      0     0    0
  15    1     1      0      3      0     0    0
  15    1     1      0      3      0     0    0
IN(Total) Q-sizes      Q-Matches    OUT
bytes act posted unexp posted unexp bytes act
  15    1     1      0      3      0     0    0
  15    1     1      0      3      0     0    0
  15    1     1      0      3      0     0    0
  15    1     1      0      3      0     0    0
  15    1     1      0      3      0     0    0
  15    1     1      0      3      0     0    0
```

MPI PERUSE is a profiling interface integrated into the Open MPI code base [14, 15]. The Sun HPC ClusterTools 8 release will be the first ClusterTools release with the MPI PERUSE interfaces compiled in. Moreover, as described previously, the DTrace functionality above was able to leverage the MPI PERUSE infrastructure in the Open MPI code base to more readily implement DTrace providers.

ZIH, TU Dresden has recently integrated VampirTrace functionality into the Open MPI code base [16]. VampirTrace can be used to output traces in Open Trace Format (OTF). Sun HPC ClusterTools 8 will include these VampirTrace traces.

Sun Studio Analyzer can be used to analyze MPI programs. The Sun HPC ClusterTools team has added support for MPI states, which can be used by Analyzer to visualize the progress of the processes in an MPI program.

4 Sun Product Activity

4.1 Installation

Sun HPC ClusterTools includes Solaris packages and a set of utilities for easily installing all the ClusterTools packages on a cluster of nodes. Once the installation is complete, all the ClusterTools software will appear in `/opt/SUNWhpc/HPC`<release number>, e.g. `/opt/SUNWhpc/HPC7.0`. The user can have multiple versions of ClusterTools installed with each installation varying the target directory based on release number. There is a tool `ctact` which allows the user to activate a particular release number, resulting in symbolic links being created in `/opt/SUNWhpc` to the appropriate directories in `/opt/SUNWhpc/HPC`<release number>.

The `ctinstall` script enables installation of the Sun HPC ClusterTools software locally on each node or in a single NFS location with symbolic links created on each node pointing to the NFS location. On a cluster of nodes, `ctinstall` provides a convenience in that the software can be installed on all the nodes in a single command.

As the Solaris packaging system evolves, so will the ability to install Sun HPC ClusterTools packages efficiently. Similarly, with support on Linux, Sun HPC ClusterTools will be able to leverage existing installation technologies such as ROCKS [17].

4.2 MPI Testing Tool

High quality MPI implementations are software packages so large and complex that automated testing is required to effectively develop and maintain them [18]. 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 frame-

work is required to: obtain and build the software to test; obtain 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.

With Cisco and Indiana University, 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. MTT has enabled us to track regressions on our Solaris clusters on a nightly basis. Moreover, we have extended MTT to support developer and release engineering environments for building and installing Sun HPC ClusterTools.

4.3 Documentation

Sun HPC ClusterTools includes a set of user documentation available as a downloadable `tar`-file or online as `html`- or `pdf`- files. The documentation set includes the following:

- *Sun HPC ClusterTools Software Migration Guide: collection of hints for users migrating from Sun HPC ClusterTools 6 to Sun HPC ClusterTools 7 and beyond.*
- *Sun HPC ClusterTools Software Installation Guide: description of how to install the ClusterTools software referenced in Section 4.1.*
- *Sun HPC ClusterTools Software User's Guide: basic usage of ClusterTools and the primary commands for compiling and running a parallel MPI job.*
- *Sun HPC ClusterTools Software Release Notes: a summary of new features in the latest release and a compendium of significant defects that the user is likely to experience.*

In future releases, there are plans to create an administrator's guide, an MPI programming guide, and a performance guide. Sun has also adapted its MPI man pages for Open MPI and contributed those to the community.

4.4 Third party support

Sun HPC ClusterTools supports both the Totalview (by Totalview Technologies) and DDT (by Allinea) parallel debuggers [19, 20]. To ensure that these parallel debuggers work properly with Open MPI, Sun worked closely with the community and these third-party vendors. In particular, Open MPI has hooks to supply the appropriate symbols to the parallel debuggers. Additionally, Open MPI includes additional functionality which provides information about MPI message queues to the parallel debuggers to view the states of these queues.

Sun HPC ClusterTools also supports PBS Professional (by Altair) [21]. Although Open MPI supports a number of resource managers, Sun does not officially support all of them. Sun HPC ClusterTools supported PBS Pro before joining Open MPI and continues to do so. Although Sun does not officially support all the resource managers, Sun works with its customers to provide a comprehensive HPC stack, even if some of those components are not provided by Sun.

5 Pros and Cons

This section describes some of the pros and cons of using ClusterTools 7 vs. ClusterTools 6.

5.1 Pros

Leveraging the community. With a proprietary MPI implementation, it would be costly to remain competitive in all state-of-the-art features and technologies. In a community, we can contribute Sun-focused features and support with advice on ways to improve those Sun features. Conversely, we benefit from the many important new features developed by the research community. For example, the Open MPI community benefits from research by Indiana University, University of Tennessee, University of Houston, and HLRS / University of Stuttgart in areas such as fault tolerance, checkpoint / restart, and collectives.

Clean-slate architecture. The clean-slate approach of Open MPI mirrors what we would have needed to do with our proprietary implementation to address the increased scalability required not only by the top-tier users but also by the volume users as MPI becomes more prevalent. The modular architecture of Open MPI is well-suited for the flexibility needed to address the diversity of users of MPI today. That is, the ability to plug-in efficient modules for communication and resource management enables the core MPI implementation to be adaptable to the preferences of a wide range of users.

No system-level administration. The architecture for our proprietary ClusterTools implementation included system-level daemons to support running MPI applications. If you wanted to run the proprietary ClusterTools, you would need root access or use a shared ClusterTools run-time environment. With resource managers nowadays much more full-featured and assuming most of the responsibilities formerly managed by the ClusterTools system-level daemons, it is less important to have system-level control and more important, at least for developers, to be able to quickly install and debug a custom self-modified version of the MPI libraries and the run-time environment totally in user space. Furthermore, there is no dependence on a system administrator for maintenance or to experience system outages caused by other users.

5.2 Cons

Robustness. As with any sophisticated software, time and use is needed to shake out the most critical issues. The Open MPI source-code is only a couple years old as compared to our proprietary ClusterTools, which had ten years of hardening through customer use. However, with a large active community and the experience of the community members, the hope is that the maturity process will happen much more rapidly than with the proprietary ClusterTools.

No system-level administration. There are always some users who do have root access and like to have complete control over their MPI jobs and run-time environment. Also, the run-time environment of the proprietary ClusterTools product had some important utilities that can be covered by resource managers, but maybe are not as focused on the MPI environment as much as an integrated run-time environment. The Open MPI community continues to develop analogous utilities and we will also provide utilities to the community and our customers as requirements dictate.

Synchronization of Sun platforms. The inherent difficulty with the community-based approach is that the community (and the state-of-the-art) are moving fast and furious and not necessarily focused on testing every platform. So if you are a platform owner, it is imperative to track regressions on your platform since others may not realize. The bottom line here is be prepared to dedicate some resources to track community progress as well as identify and fix issues quickly.

6 Future work and conclusions

For future work, we will draw upon our past experiences and positive attributes as identified by customers of the proprietary ClusterTools product. For other areas of future work, we plan to focus on areas that the marketplace dictates.

Scalability, performance, and robustness remain primary competitive differentiators. Scalability of job startup and collective operations on large clusters have become as fundamentally necessary as latency and bandwidth performance of basic communication. When threading is involved in such super-scalable jobs, increased levels of thread safety will be required by users. In jobs with so many processes, the robustness of the MPI implementation becomes more important than ever and better yet the ability to gracefully react to failures of individual nodes and/or processes. Processor affinity also remains important to aid in efficient and scalable shared memory performance.

Beyond continually striving for world-class scalability and performance, increasing the ease-of-use for customers, as with any MPI implementation, is very important to help those customers who do not wish to become versed on the subtle details of the MPI specification and/or the details of a particular MPI implementation.

With insatiable appetite for greater scale and performance and increasing requirements for reliability and persistence through failures, it seems such lofty goals are best-suited for a community approach. Hence, our participation in the Open MPI community and adoption of the Open MPI open-source code base as the basis for our MPI product will attempt to leverage the state-of-the-art work of our fellow community members while we contribute our experience as well in addition to our product-oriented focus.

References

1. Sun HPC ClusterTools web page, http://www.sun.com/clustertools
2. Open MPI references, http://www.open-mpi.org/papers
3. Open MPI web page, http://www.open-mpi.org
4. Sun Studio web page, http://developers.sun.com/sunstudio
5. Sun Grid Engine web page, http://www.sun.com/software/gridware
6. Grid Engine web site, http://gridengine.sunsource.net
7. DTrace web page, http://www.sun.com/bigadmin/content/dtrace
8. MTT web page, https://svn.open-mpi.org/trac/mtt
9. Sistare, S., D. Allen, R. Bowker, K. Jourdenais, J. Simons and R. Title (1994), A Scalable Debugger for Massively Parallel Message-Passing Programs, IEEE Concurrency, Vol. 2, No. 2.
10. OpenSolaris web site, http://www.opensolaris.org
11. OpenFabrics Alliance web site, http://www.openfabrics.org
12. DAT collaborative uDAPL web page, http://www.datcollaborative.org/udapl.html
13. Using ssh with qrsh and qlogin, http://gridengine.sunsource.net/howto/qrsh_qlogin_ssh.html
14. MPI PERUSE web site, http://www.mpi-peruse.org
15. Keller, R., Bosilca, G., Fagg, G., Resch, M. M., Dongarra, J. J. (2006), Implementation and Usage of the PERUSE-Interface in Open MPI, LNCS 4192:347–355.
16. VampirTrace FAQ for Open MPI, http://www.open-mpi.org/faq/?category=vampirtrace
17. ROCKS web site, http://www.rocksclusters.org
18. Hursey, Josh, E. Mallove, J. Squyres and A. Lumsdaine (2007), An Extensible Framework for Distributed Testing of MPI Implementations, Euro PVM/MPI 07.

19. Totalview Technologies web site, http://www.totalviewtech.com
20. Allinea web site, http://www.allinea.com
21. Altair web site, http://www.altair.com.

An Integrated Environment For the Development of Parallel Applications

Gregory R. Watson and Craig E. Rasmussen

Abstract The development of parallel applications is becoming increasingly important to a broad range of industries. Traditionally, parallel programming was a niche area that was primarily exploited by scientists trying to model extremely complicated physical phenomenon. It is becoming increasingly clear, however, that continued hardware performance improvements through clock scaling and feature-size reduction are simply not going to be achievable for much longer. The hardware vendor's approach to addressing this issue is to employ parallelism through multiprocessor and multi-core technologies. While there is little doubt that this approach produces scaling improvements, there are still many significant hurdles to be overcome before parallelism can be employed as a general replacement to more traditional programming techniques. The Parallel Tools Platform (PTP) Project was created in 2005 in an attempt to provide developers with new tools aimed at addressing some of the parallel development issues. Since then, the introduction of a new generation of peta-scale and many-core systems has highlighted the need for such a platform. We describe the current state of PTP, and discuss how a new generation of tools is going to be required to meet the needs of these architectures.

1 Introduction

Parallel computers have existed in one form or another almost since the first computers were available. The complexity introduced by parallelism was evident from a very early stage, and has been a major impediment to the adoption of paral-

Gregory R. Watson
IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, e-mail: grw@us.ibm.com

Craig E. Rasmussen
Los Alamos National Laboratory, Bikini Atoll Road, Los Alamos, NM 87594, e-mail: crasmussen@lanl.gov

lelism in main stream application development. Many programming models and techniques have been used to improve the simplicity and reliability of parallel programs. Dozens of new languages and language features were introduced, however very few are still widely used. In the 1990's, the Message Passing Interface (MPI) standardization effort [12] was seen as a major step forward in parallel programming models. The predominant programming models still in use are asynchronous threads and MPI, although the use of partitioned global address space (PGAS) languages, such as Unified Parallel C (UPC) [16], appear to be increasing in popularity. Although the PGAS languages simplify the programmer's task to some extent, the potential for deadlocks and other synchronization issues still remain a significant challenge.

The first integrated development environment (IDE) was introduced when computer input devices became sophisticated enough to support the seamless integration of development activities. Due to performance and usability issues, however, there was often programmer resistance to the wholesale adoption of IDEs. The quality and productivity improvements achieved using IDEs is now well established [6, 7, 10, 13]. Combined with improvements to the IDEs themselves, this has now resulted in IDEs being the predominant environment for software development. Although a considerable number of IDEs are available today, many are limited to a single operating system (e.g. KDevelop, Visual Studio), or are proprietary (e.g. Visual Studio, Xcode, Sun Studio). Eclipse is one of the few truly cross-platform IDEs that has been designed for extensibility. Interestingly, although IDEs have been used in the past to aid parallel application development [2, 3, 5, 9], none of these are still available today. Few developers working on parallel scientific codes use IDEs at all.

The Eclipse Parallel Tools Platform (PTP) was launched in 2005 in an attempt to address this situation. At this time, beowulf-style clusters had largely replaced custom proprietary parallel hardware for high performance computing (HPC), however the predominant parallel application development environment was still command-line tools. At the same time, the move towards multi-core architectures for conventional applications was outpacing the ability of existing IDEs to provide the tools necessary to exploit the new technology. As the HPC and conventional architectures begin to converge, the need for sophisticated tools and new programming models has become even more urgent.

PTP builds on the exemplary tools available in the Eclipse platform and the C/C++ Development Environment (CDT) to provide support for C, C++, UPC, Fortran, and in the future other parallel languages. It is also a *platform*, so that while it provides a range of core services and tools, it is also designed to be extended to support new tools, architectures, and programming models. In addition to Eclipse's advanced editing, build, and integrated source code management functionality, PTP provides four additional features: advanced error checking and analysis tools that assist the programmer to develop parallel applications; runtime monitoring and control of parallel jobs; debugging support for multi-process applications; and a performance tools framework for the integration of parallel performance tools. In the first of these, PTP provides a number of tools that are primarily aimed at the MPI and OpenMP [4] programmer, and that reduce much of the tedious and error-prone

nature of these programming models. Runtime monitoring and control of parallel jobs abstracts the interaction between the developer and the parallel system, so that the developer is able to seamlessly launch and control applications without needing to focus on specific architecture details. The debugging support provides a parallel debugging platform with basic debug functionality, but that can be extended to encompass the new debugging paradigms that will be required on peta-scale and multi-core systems. The performance tools framework allows existing performance tools to be easily integrated into the Eclipse framework so they are accessible to the developer.

In the following sections, we will outline some of the challenges faced by developers and our approach to overcoming these, the architecture and major features of PTP, and future directions for the PTP project.

2 Challenges

With the growing popularity of multi-core systems as a means of improving application performance, parallel programming is set to enter the main stream. Although threads have been used effectively as the predominant programming model for shared memory architectures, explicit threading is neither easy to program correctly, nor conducive to retrofitting applications in order to utilize the new architectures. How existing applications will benefit from the new age of parallelism without huge investments in reengineering is still very much an unanswered question.

In scientific computing, explicit parallelism has been employed with varying degrees of success for many years. Unfortunately, the homogeneous architectures that have facilitated these programming models have reached a practical limit in the search for peta-scale performance and beyond. One approach to addressing this is to offload large portions of the computation load onto some form of accelerated hardware. The result is a very heterogeneous environment that introduces significant complexity into the application development process. In an attempt to address these problems, a large scale effort is underway to develop new programming models and languages that will reduce the complex and error-prone nature of parallel application development, and to develop new tools that will aid both legacy and new applications to extract the maximum performance from the new architectures. We will not focus on the issues facing scientific application developers any further, as this has been addressed in detail elsewhere [8].

Since the use of IDEs is predominant across the computing industry, it is not unreasonable to expect this to continue as the adoption of parallel architectures becomes more widespread. Whether the scientific developer community adopts IDEs in a wholesale manner still remains an open question, but it is the opinion of the authors that this will be an inevitable result of the complexity of the new platforms. IDEs like Microsoft's Visual Studio, Sun's Studio One, Apple's Xcode, Eclipse, and others will need to be adapted to support these platforms, and the languages and programming models that they encompass. Further, the tools that will be required

to extract optimal performance will also need to be integrated so that they form a seamless part of the development lifecycle.

The challenges facing the IDE developer in adapting to this changing landscape are numerous. Currently most IDEs make a number of assumptions about the environment. These include:

- The IDE runs on same platform as the development environment (embedded systems are a notable exception)
- The developer has exclusive access to resources for development purposes
- Platform parallelism is handled by the operating system (threads/SMP)
- The development toolchain is simple (single pass)
- Optimized performance can be achieved by the compiler, or by manual reasoning about behavioral characteristics of the program
- Languages will continue to be text-based
- The number of executing tasks is relatively small

In the future, many, if not all, of these assumptions will change. In scientific computing it is already unusual for computational resources to be available locally, and development environments are becoming complex enough to require significant resources in themselves (e.g. building large applications can take many hours). In these environments, the ability to develop applications remotely will be an important requirement. Large-scale multi-core systems are likely to require similar remote development capabilities.

Another assumption that is likely to change is that threading models will continue to be the predominant paradigm, and hence that parallelism will be managed transparently by the operating system. The experience from scientific computing is that parallel applications require significantly more infrastructure than can be provided by the operating system alone. This has the effect of complicating the build model (requiring additional libraries, etc.), the runtime environment (applications can no longer be run by simply launching a single executable), and application debugging. Most tool chains used to build parallel applications currently assume that the process is a linear sequence of compile and link steps. However, as architectures become more complex, it is possible that many more activities will be required to produce an optimized application. For example, multiple programming models may be combined (as is already required for IBM's Cell Broadband Engine), or information gathered at runtime may be required to augment the static analysis performed by the compiler.

The DARPA HPCS Language Project [11], an attempt by DARPA to improve software development productivity, has resulted in at least one parallel language that is no longer strictly text-based [1]. It may also become necessary to break this link with traditional text-based languages in order to provide access to new language features that are precluded by textual representation (visual programming is one such example.)

The final assumption is also changing swiftly, with peta-scale machines expecting in the order of 1M executing tasks, and existing threaded applications, which already exhibit thousands of threads, are likely to also increase in size significantly.

Dealing with large numbers of objects (threads, processes, etc.) raises many scalability issues, both in the ability of the IDEs user interface to display and manage the objects, and in communication services that are used between remote systems and the local environment.

All these assumptions can have a profound influence on the architecture and functionality of an IDE, but there are also additional challenges. As programming models evolve, and new languages are developed, the IDE needs to be able to adapt without a significant re-engineering effort. This also applies to the new types of hardware and systems that are currently under development. It is also clear that the tools required for parallel programming are going to have to be significantly more powerful that those available today. In particular, it is likely that static analysis of programs and refactoring will play an important part in making parallel programming more widely acceptable. In order to support these types of tools, an IDE must provide the necessary infrastructure to make this possible. Such infrastructure is decidedly more complex than that required for simple syntax highlighting or providing an outline view of the program, or that is typically available in editors such as Emacs.

From our early analysis of existing IDEs in 2005, there was only one that came close to meeting the criteria for a parallel development environment, and that is what we used for the basis of PTP. Of course, not all the issues have yet to be addressed, but the flexibility and extensibility of Eclipse will ensure that PTP will be able to evolve to support the demands of future parallel application developers.

3 Architecture

The Parallel Tools Platform is an extension to the Eclipse platform that fulfills three main goals: provide the tools and infrastructure necessary for advanced error checking and analysis of parallel applications; provide a runtime environment that allows developers greater transparency into the systems on which they are developing applications; and provide a debugger that will allow developers the ability to more easily locate errors and anomalies in program behavior. In the following sections we will describe each of these aspects of PTP in more detail.

3.1 Analysis Tools

The PTP analysis tools are aimed at providing Eclipse with an additional feature set that is designed to aid the development of parallel applications. These tools are currently targeted at the MPI and OpenMP programming models, but we fully expect them to be extended to other models or languages as the need arises.

3.1.1 Advanced Help and Content Assist

Eclipse includes an integrated help system that provides a help browser and context sensitive help that can be accessed directly from the user's editor session. PTP augments this help system with MPI- and OpenMP-specific information in order to improve the developer experience when using these programming models. Reference information about the MPI and OpenMP API, including arguments, return type, and a description, are available via the help browser or by simply placing the cursor over an API in the editor view to activate hover help. The Eclipse content assist has also been augmented to enable auto completion of APIs and arguments while typing.

3.1.2 Artifact Analysis

This analysis tool allows the developer to more easily work with MPI and OpenMP codes by providing a higher level abstraction of the APIs. Like the *outline view*[1], the *artifact view* shows a list of all MPI function calls, Open MPI pragmas, and other artifacts in the program. Figure 1 shows the MPI artifact view. Navigation to the source code location of these artifacts is achieved by clicking on the artifact in the view, or by using the icons in the navigation bar.

In addition to augmented views, the artifact analysis also provides more advanced error checking features than are typically available in Eclipse. These are the types of checks that could be made by compilers, but by providing an integrated tool it is possible to provide error reporting much earlier in the development cycle. Currently, checks for many of the known OpenMP programming errors are provided.

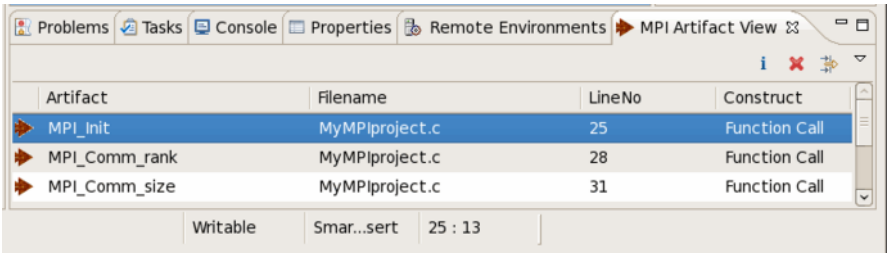


Fig. 1 View showing MPI artifacts discovered in the source code

[1] The outline view provides an outline of the program showing its structural elements.

3.1.3 Barrier Analysis

The barrier analysis tool can be used to detect potential deadlocks in MPI applications. The tool does this by identifying the location of all MPI barrier statements[2] in the application (which may be scattered throughout the source code), and constructs *barrier matching sets*. Each set comprises all the barrier statements that could execute in parallel. Using this information, it is possible to determine if there are any barrier statements that do not have a matching barrier, and flag these as potential deadlock errors. In addition, a barrier view is provided to enable the easy navigation to barrier statements in the source code. Figure 2 shows an example of the barrier view containing a list of barrier sets.

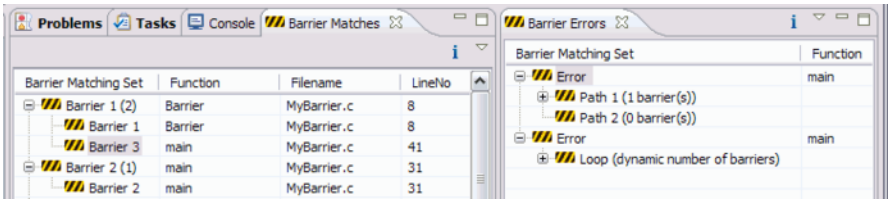


Fig. 2 View showing barrier matching sets and barrier errors that were discovered using static analysis

3.1.4 Concurrency Analysis

Like the barrier analysis tool, the concurrency analysis tool is used to detect potential concurrency problems, but for OpenMP (threaded) applications. The concurrency analysis tool allows the developer to choose a particular expression, and will evaluate and identify all expressions that could execute concurrently with the selected expression. Since it is important to ensure that only expected expressions execute in parallel, this tool can be used to detect potential race and deadlock conditions.

3.2 Runtime Tools

One of the difficulties facing the parallel application developer is the lack of transparency about the behavior and status of applications and the machines that they run on. Further, many parallel systems have a more complex interface than POSIX-style execution, and because they are a scarce resource, typically employ a job scheduler

[2] An MPI barrier causes each process to wait until all processes have reached a barrier. It is used to synchronize all processes.

to manage access to the computational resources. Not only must the developer spend time learning the interfaces and integrating these with their development processes, but each iteration of the development cycle can be encumbered with unnecessary and tedious activities.

To facilitate a more productive development environment, PTP provides a number of abstractions that simplify the interaction with target systems. The first of these is a *runtime model*[3] that provides an abstract representation of the parallel system that the developer is interacting with. This model forms the core of a model-view-controller design pattern around which PTP is based. Information about the parallel system, and applications running on the system, is fed into the model in the form of events which update the status of model elements. PTP provides a number of views into this model that enable the developer to monitor the status of the system and the applications as they are executing.

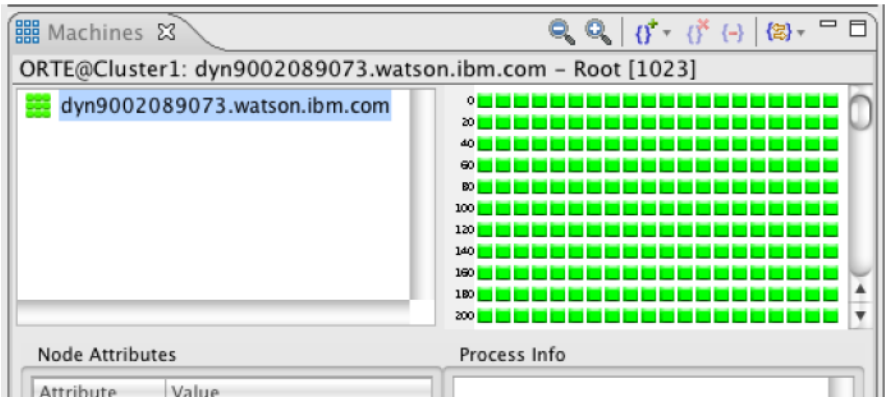


Fig. 3 View showing the status of the first 220 nodes of a 1024 node cluster

The second abstraction that PTP provides is the notion of a *resource manager*, which represents any subsystem that manages resources on a target system. Examples of resource managers include: MPI runtime systems; job schedulers; virtual machines; and simulators. PTP allows multiple resource managers to be configured, and places no restrictions on the location of the resources, so they can be local or remote to the Eclipse environment. Internally, a resource manager is just another part of the runtime model hierarchy, so the model views can be used to provide a display of the status of any resource managers that have been configured. Interaction with remote resource managers is achieved using a small proxy agent that is started on the remote system using one of Eclipse's built-in remote service providers. Communication with this agent can be tunneled over a secure ssh connection to address the security requirements of many installations. In addition to monitoring activities, the agent is also used to control resource manager operation, submit jobs for execution,

[3] Not to be confused with a programming model. The runtime model only provides a model of the parallel machine for monitoring and control purposes

and initiate debug sessions. Figure 3 shows an example of the *machines view* for a 1024 node cluster.

Launching of parallel applications is managed through the normal Eclipse launch configuration mechanism. PTP adds a parallel application launch type that allows the developer to select the resource manager that will be used to control job submission, and supply resource manager specific attributes that specify resource constraints on the job. Once a job has been submitted, the runtime views allow the user to monitor progress of the job on the target system. Figure 4 shows an example of the *jobs view* with a selection of jobs in various states.

Fig. 4 View showing four jobs in various states (red - completed, green - running), and the 64 processes in `job05`

3.3 Debug Tools

A key aspect of any development process is the ability to effectively locate and correct program errors. Debugging has traditionally been a difficult area for parallel application developers, since traditional debugging methodologies only apply when the number of parallel tasks remains small, and the very act of debugging can perturb the application enough to make identifying temporal issues very difficult. Very few parallel debuggers currently exist, so developers have, until recently, only had a relatively few options:

- Purchase a commercial parallel debugger
- Attempt to use a sequential debugger (such as `gdb`) or a debugger wrapper (such as `mpigdb`)
- Use debug print statements (`printf` or equivalent)

As only a small number of commercial parallel debuggers exist[4], there is little competition to drive innovation and new functionality, and with only a small potential market, this can be an expensive debug solution. Also, these debuggers suffer from scalability problems when debugging applications larger than a few thousand processes. The `gdb` or `mpigdb` options, while cheaper, also suffer from scalability and usability issues. Neither the commercial nor open source solutions are integrated with a complete development environment, so launching a debug session can

[4] At the time of writing only two: TotalView and DDT.

be a challenging exercise. Using debug print statements, while neither scalable nor powerful, is at least ubiquitous and easy to use. As a result, this has become the de facto debugging paradigm for parallel programming.

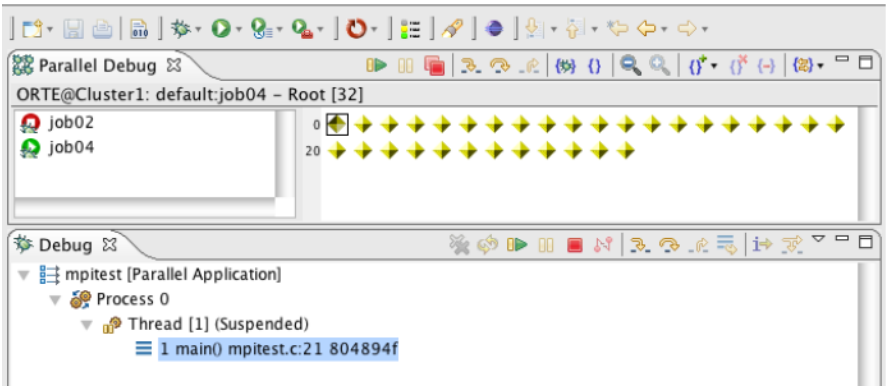


Fig. 5 Parallel debug view showing a 32 process job being debugged

PTP attempts to overcome these limitations, by providing an integrated parallel debugger that can be activated whenever the developer requires detailed debugging information about the application under development. In addition to normal debugging functionality, such as setting breakpoints, single stepping, viewing and altering variables, etc., the debugger also gives the developer the ability to control and manipulate arbitrary sets of processes associated with a parallel application as it is executing. By default, the debugger establishes a set of all processes in the application run, and commands such as setting a breakpoint, single stepping, or resuming execution can be applied to this set of processes. The set can be subdivided into an arbitrary number of subsets (including individual processes) that allow finer control of application execution. Figure 5 shows the *parallel debug view* which allows manipulation of sets of processes.

Debugger scalability is always an issue, and the PTP debugger is no exception. However, the debugger infrastructure has been designed to scale, and so far has proved effective up to the same application sizes that can be handled by the commercial debuggers. In addition, because the PTP debugger is an open architecture, we hope that it will be used as a platform to develop new debugging paradigms that will be necessary to deal with applications that comprise hundreds of thousands or millions of parallel tasks.

4 A Simple Case Study

In the following section, we will present a simple case study on using PTP for developing an MPI application. This will include describing the steps necessary to

import and configure an existing MPI application in PTP, locate a potential deadlock situation, then launch the application under debugger control. The steps are as follows:

1. Import the source files into an Eclipse-controlled project
2. Configure the project to correctly locate external tools (e.g. the `mpicc` command)
3. Run the barrier analysis on the source code, and correct any potential deadlock errors
4. Build the executable
5. Configure a launch configuration
6. Start a debug session

4.1 Importing

Eclipse offers a range of options for importing an existing application so that it can be developed using PTP. We will describe the three main types here.

- Copying into the workspace. This first option allows an existing project to be copied into the Eclipse workspace. It is useful if the developer wishes to keep the original source files pristine, or if Eclipse will be used as the primary development environment.
- Linking to an external project. This option allows an existing project to be used in Eclipse, but without disturbing the location or layout of the files. Eclipse creates an internal link to the project, so that the project files appear in the user interface.
- Checking out from a source code repository. This option allows a copy of project controlled by a source code repository[5] to be checked out into the local workspace. Modifications to files are automatically detected by Eclipse, and the developer can perform operations such as committing changes, comparing versions, merging, branching, etc.

The developer simply selects the import method they desire, and imports the source code into the Eclipse workspace. Eclipse is scalable enough to support very large projects (thousands of files, millions of lines of code). Activities such as indexing the source code (used for advanced searching, content assist, type and call hierarchy views), are potentially long running and automatically take place in the background without affecting the developer.

[5] Eclipse supports CVS, SVN, and other repositories.

4.2 Configuring

Projects controlled by Eclipse have a large number of configurable options. Since this application is using MPI, we require it to be compiled and linked with the `mpicc` command. Also, the MPI analysis tools need to know the location of the MPI header file `mpi.h`, so this also has to be added to the project configuration. In Eclipse, building a project is controlled by a *toolchain*[6]. Both the compiler name and the include path are set by modifying the toolchain options for the project.

4.3 Analyzing

The barrier analysis tool is invoked on the project using a special menu on the Eclipse *toolbar*[7] . The analysis will scan all source code in the project and compute the barrier sets. Markers indicating the location of potential errors will be placed on corresponding source files and when the source file is opened, at the source line location in the file. The developer can now use this information to correct the deadlock situation.

4.4 Building

Eclipse projects can be configured to automatically build each time an editor change is saved, or by manually invoking the build command from an Eclipse menu. While the build is running, the developer is able to continue to modify the source, perform analysis, or undertake other activities that are not dependent on the build completing. Build progress is displayed in a special *Progress view*, that provides an estimate of the percentage completed. Detailed output from the build is available in the *Console view*. If any errors are detected by the compiler or linker steps, the build will terminate, and a list of the errors will be displayed in the *Problems view*. Markers will also be placed on source files and displayed in the editor.

4.5 Launching

Once the build is complete, the developer must configure a launch configuration to run (and debug) the application. A single configuration is used for both running and

[6] A toolchain describes the sequence of commands required to convert the source code into a binary executable.

[7] The toolbar provides quick access to commonly used functions via a series of icons at the top of the Eclipse window.

debugging. The launch configuration specifies the attributes needed to launch the application, such as the executable name, command line arguments, environment variables, etc. These attributes are saved in the configuration, so they only need to be specified once. After creating the configuration, the application can be run or debugged by clicking a single button on the toolbar.

4.6 Debugging

When the developer is ready to debug the application, a single button click will invoke the debugger. Eclipse will automatically switch to display views for controlling the application (e.g., single stepping), examining stack frame location, viewing variables, etc. Breakpoints can be set directly in the source code editor view by clicking on the left edge of the view. Once the debug session is completed, the developer can switch back to the runtime and editor views with a single click.

5 Future Directions

There are many aspects of parallel application development for both peta-scale and the emerging multi-core systems that still remain a major challenge. The current programming models are unlikely to be adequate for applications designed to run on peta-scale systems, and much more powerful tools will be required to optimize performance for the next generation of heterogeneous hardware. If multi-core systems are going to become the performance panacea, then application developers will need programming models and languages that are as simple and easy to understand as those being used today. Eclipse and PTP are well placed to assist with both these environments. In the following sections, we briefly examine areas where future development of PTP appears promising.

5.1 Analysis Tools

There are a number of tools available that provide analysis information that can be derived from running the application, such as trace and profile information, and that could be used to augment static analysis and provide greater insights into program operation. In addition, there are opportunities to better utilize compiler generated information to assist in the application development process. One such tool being actively developed will use compiler generated parallelization analysis to aid the developer in parallelizing selected code regions.

5.2 Performance Tools

PTP provides a performance tools framework for integrating performance tools with Eclipse, however this is only a small part of the functionality required to support integrated performance analysis and optimization of parallel applications. Ideally, the developer should be able to invoke a performance analysis tool as easily as launching or debugging the application, have the data automatically collected and analyzed, and the results used to annotate the source code. The Tuning and Analysis Utilities (TAU) have already been integrated with PTP, and a number of other performance tools groups are also exploring Eclipse as a delivery platform. However, there is still much work to do to ensure that performance tools can be easily and effectively used as part of the development workflow.

5.3 Multi-core Tools

The current PTP tool set has been targeted primarily at distributed memory architectures and programming models (with the exception of OpenMP), however there is a growing requirement for tools to ease the transition from existing architectures to multi-core systems. At least three kinds of tools could benefit these applications: tools to aid in parallelizing sequential applications in order to make better use of the increased compute resources; performance analysis tools specifically targeting applications running on multi-core systems; and debugging tools that better manage the extra complexity introduced by multi-core architectures.

5.4 New Languages and Programming Models

A variety of efforts are underway to develop new languages and programming models for parallel computing. In addition to the DARPA HPCS Language Project, there are also projects aimed at enhancing existing languages, such as UPC, Co-Array Fortran (CAF) [14], and Titanium [18], that add new functionality to better support parallel programming. New programming models, such as Asynchronous Partitioned Global Address Space (APGAS), on which IBM's X10 language [17] is based, are being developed. There is considerable scope for adding support for these languages and programming models to PTP.

5.5 New Debugging Methodologies

The existing interactive debugging methodology for parallel applications is not significantly different from that used for sequential applications. However, as the size

of applications increases to peta-scale and beyond, it is not clear that this methodology will remain effective. In particular, if applications comprise millions of concurrently executing tasks, just identifying which tasks are the source of the errors is likely to become a highly challenging activity. The rich user interface and extensibility of Eclipse provides an exciting opportunity to investigate new techniques for analyzing, locating, and correcting errors in parallel programs.

6 Conclusion

The quest for greater hardware performance is driving a significant change in the application development landscape. Both the scientific and mainstream computing communities are facing the challenge of developing parallel applications that are able to extract maximum performance from the new hardware. There is no doubt that new tools, languages, and programming models will be needed to assist the developer to reach this goal.

Although a number of integrated parallel tool environments have been developed in the past, none are still in wide use today. It's possible to speculate on the reasons for this, but one factor is clear: none have been based on a framework that enjoys the enormous popularity and the advanced features of the Eclipse platform. In addition to an open, portable and robust platform, Eclipse also provides an extensive array of advanced tooling that can be used by tool developers to create an integrated solution to a wide array of programming activities.The Parallel Tools Platform builds on this solid foundation, and provides an additional framework for developing and integrating tools for developing parallel applications. Currently, PTP provides a range of tools that provide advanced error checking, static analysis, runtime monitoring and control, and debugging services.

In addition to the existing tools, there are a number of efforts underway to improve the range of tools and functionality that PTP provides. This includes extending the analysis support to encompass dynamic analysis, and better integration for performance analysis tools. There are also active projects to enhance the ability of Eclipse to work in distributed development environments, and to improve the refactoring support that is available for existing programming languages.

PTP is still a very young project, and there are many opportunities for improving the capabilities to suit the advances in computing technology that will be introduced over the next few years. The integrated nature of the platform also offers scope for developing new tools, that may have not been possible in the past, to deal with programming challenges that will be faced by both the peta-scale and many-core communities.

Acknowledgements The authors would like to acknowledge the efforts of many contributors without whom the Parallel Tools Platform would not exist. This includes the Eclipse Foundation, Los Alamos National Laboratory, Monash University, IBM Corporation, University of Oregon, Oak Ridge National Laboratory, and Technische Universität München, along with the many in-

dividuals who have shared their ideas and suggestions. Thanks also to Beth Tibbitts for various images used in the document.

References

1. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, G. Steele, S. Ryu, S. Tobin-Hochstadt: The Fortress Language Specification. Available via http://research.sun.com/projects/plrg/
2. B. Q. Brode, C. R. Warber: DEEP: A Development Environment For Parallel Programs. Proceedings of the International Parallel Processing Symposium, 1998, pp. 588–593
3. D. Callahan, K. Cooper, R. Hood, K. Kennedy, L. Torczon: ParaScope: A Parallel Programming Environment. Proceedings of the First International Conference on Supercomputing, Athens, Greece, June 1987
4. R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald: Parallel Programming in OpenMP. Morgan Kaufmann, 2000.
5. J. Cownie, A. Dunlop, S. Hellberg, A. J. G. Hey, D. Pritchard: Portable Parallel Programming Environments - The ESPRIT PPPE Project. Massively Parallel Processing Applications and Development, Netherlands, June 1994
6. A. Frazer: CASE and its Contribution to Quality. The Institution of Electrical Engineers, London, 1993.
7. M. J. Granger, R. A. Pick: Computer-aided Software Engineering's Impact on the Software Development Process: An Experiment. Proceedings of the 24th Hawaii International Conference on System Sciences, January 1991, pp. 28–35
8. L. Hockstein, V. R. Basili: The ASC-Alliance Projects: A Case Study of Large-Scale Parallel Scientific Code Development. IEEE Computer, Vol. 41, No. 3, March 2008, pp. 50–58
9. P. Kacsuk, J. C. Cunha, G. Dózsa, J. Lourenço, et. al.: A Graphical Development and Debugging Environment for Parallel Programs. Parallel Computing Vol. 22, No. 13, February 1997, pp. 1747–1770
10. P. H. Luckey, R. M. Pittman: Improving Software Quality Utilizing an Integrated CASE Environment. Proceedings of the IEEE National Aerospace and Electronics Conference, May 1991, pp. 665–671
11. E. Lusk, K. Yelick: Languages for High-Productivity Computing: The DARPA HPCS Language Project. Parallel Processing Letters, Vol. 17, No. 1, pp. 89–102, 2007
12. MPI: A Message Passing Interface Standard, Message Passing Interface Forum. Available via http://www.mpi-forum.org, June 1995
13. R. J. Norman, J. F. Nunamaker Jr.: Integrated Development Environments: Technological and Behavioral Productivity Perceptions. Proceedings of the Annual Hawaii International Conference on System Sciences, January 1989, pp. 996–1003
14. R. Numrich, J. Reid: Co-Array Fortran For Parallel Programming. In ACM Fortran Forum, Vol. 17, No. 2, pp. 1–31, 1998
15. M. Snir, P. Hochschild, D. D. Frye, K. J. Gildea: The Communication Software and Parallel Environment of the IBM SP2. IBM System Journal, IBM Corp., Vol. 34, No. 2, pp. 205–221, 1995
16. UPC Language Specification v1.2. UPC Consortium, Berkeley National Laboratory, 2005
17. The X10 Programming Language. Available via http://www.research.ibm.com/x10
18. K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken: Titanium: A High-Performance Java Dialect, Concurrency: Practice and Experience. Vol. 10, pp. 825–836, 1998

Debugging MPI Programs on the Grid using g-Eclipse

Christof Klausecker, Thomas Köckerbauer, Robert Preissl, and
Dieter Kranzlmüller

Abstract With the increasing need for more computational power rises the number of processors in modern high performance computers. Coupled to the scalability of the system, the complexity of the communication between these processors increases vastly. This development can also be seen in today's Grid infrastructures, where high numbers of resources are shared over long distances. As a result, there is an intensified need for tools to support debugging and program understanding. The focus of this work is on MPI applications running on Grid sites, and we describe the corresponding functionality of g-Eclipse's developers perspective. The g-Eclipse framework is based on the open source Eclipse platform and extends its functionality by middleware independent plug-ins for Grid users, operators and developers. The provided functionality of the developer perspective includes remote building, debug support for individual MPI processes as well as graphical representation of message-passing based communication. This paper provides an overview of each of these functions and examples for their application in program development.

1 Introduction

Grids today are established as a tool for scientists to solve their scientific problems. In order to utilise Grids, developers can either use the low-level functionality of the individual Grid middleware, or work with independent development tools. Apart from tools such as the Grid Development Tools (GDT) [2], which is based on Globus Toolkit 4 (GT4) and service-oriented Grid computing there are not many integrated development environments for Grid infrastructures available. To remedy

GUP – Institute of Graphics and Parallel Processing, Joh. Kelper University Linz, Altenbergerstr. 69, A-4040 Linz, Austria/Europe, http://www.gup.jku.at/, e-mail: cklause@gup.jku.at

This paper is in parts replicated from the diploma thesis of Christof Klausecker.

this, g-Eclipse [16] [1,2] is being developed - an integrated and user-friendly middleware independent environment.

The g-Eclipse framework provides a set of tools for different Grid actors offering them three perspectives, supporting their role of either being a user, an operator or a developer, developing applications for the Grid. Among other tools the event trace and debugging functionality of g-Eclipse as described in this paper is settled in the developers perspective.

This paper presents some aspects of the g-Eclipse's developer perspective with focus on the parts dealing with MPI applications running on Grid sites. The paper is organized as follows: The next section provides an overview of related work in this domain, followed by an overview of the approach in Section 3. The individual components, the remote builder, the grid application launchers, and the trace viewer are described in Section 4-6, before a summary and an outlook on future work concludes the paper.

2 Related Work

From a chronological view point, the tools ATEMPT [3] and DeWiz [5] are seen as the predecessors of the event trace functionality in g-Eclipse. The experiences gathered with the design and usage of these tools were the main influences during the development of the trace visualization component for g-Eclipse.

Apart from the two tools mentioned above, several other similar examples exist, including Vampir [9] and Jumpshot [17] which mainly focus on performance visualisation. Other approaches providing debugging support for applications running on the grid include tools like Worqbench [7], Net-dbx-G [11] and a grid-enabled version of p2d2 [4]. The Worqbench framework allows to debug remote programs running in a Globus Toolkit 4 (GT4) environment, by adding an additional web-service to it. While it offers integration into Eclipse using its own debugging front-end, it does not provide support for executing MPI programs on the grid. Net-dbx [10] is a Java based tool allowing to debug MPI applications over the internet. With Net-dbx-G an enhanced version that supports grid environments exists. It offers a graphical user interface implemented as Java applet, thus allowing to debug applications by solely using a web browser. The p2d2 project created a client-server based debugging architecture which provides its own graphical user interface, however the tool is not publicly available.

Another related project is the Parallel Tools Project (PTP)[3], which like g-Eclipse is hosted by the Eclipse Foundation. It provides support for debugging parallel pro-

[1] g-Eclipse is a two-year project funded under the European Union's 6th Framework Programme, Contract Number IST-0343272. Since the end of October 2006, g-Eclipse is also an official Eclipse Technology Project.

[2] http://www.geclipse.eu/

[3] http://www.eclipse.org/ptp/

grams in Eclipse by offering a scalable parallel debugger. Yet, it has no explicit support for grids.

3 Overview of g-Eclipse Approach

The g-Eclipse approach consists of three major components:

- Remote Builder
- Grid Application Launchers
- Trace Viewer

The builder compiles and links programs remotely, launchers take care of running and debugging applications on the grid, and a graphical component allows to analyse the communication between parallel processes based on pre-recorded program traces.

Figure 1 shows the developed components put together into an Eclipse perspective, allowing to conveniently debug remote running MPI applications from the local desktop.

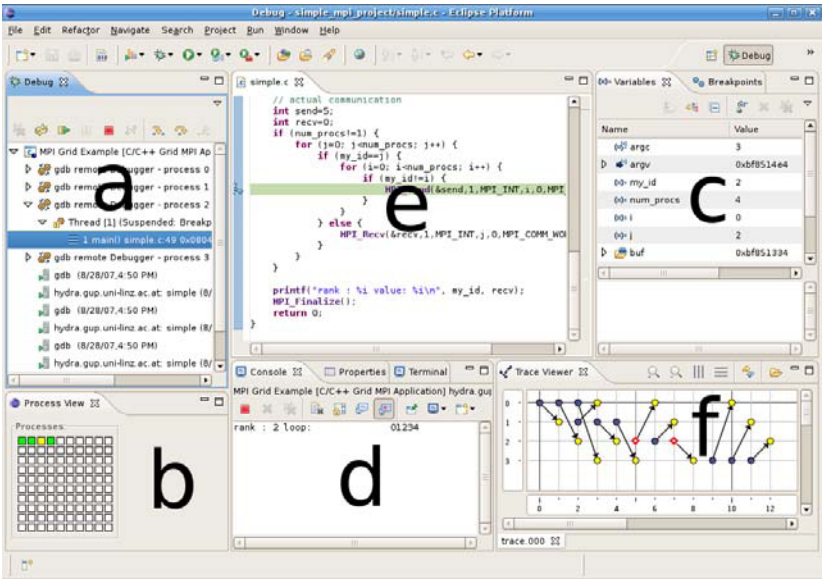


Fig. 1 Debug perspective showing the views for MPI debugging

The debug view (a) of Eclipse's standard debug perspective lists the remote running MPI processes. To provide a better overview of the processes and their current state, the so called "Process View" (b) which displays a graphically representation of each process can be used. Both of these views allow to steer the remote running

application by stepping through the program or resuming the debugee. The variables view (c) shows the content of the selected process' variables. The in- and output of the remote running application are connected to the standard Eclipse console (d), thereby providing the possibility to interact with the program. The C/C++ editor (e) provided by CDT is used to display the source code. The trace viewer (f) shows a pre-recorded trace allowing post-mortem analysis of the inter-process communication.

4 Remote Builder

An integral part of the functions of an Integrated Development Environment (IDE) is the possibility to build projects from source code after modifying the code or changing the resource where it should be executed. In the best case, this should happen in a convenient way without forcing the user to leave the environment and to enter cryptic commands on a console. A compiler and the corresponding linker have to be executed somehow to create the binary executable file. The possibility to build projects is also required during the debugging process, which is an iterative process that requires source code changes, and is often carried out in a trial and error fashion. However, running and debugging applications on a remote host introduces a series of difficulties.

The simplest solution would be to compile the application on the local machine and afterwards stage the executable to the grid. This would require a homogeneous environment, which is per definition unrealistic in the grid world. In reality not only the grid itself is heterogeneous, but the users, who develop and debug the applications, also have different operating systems with different libraries running on machines with different architectures. Furthermore specific libraries, like for MPI, are usually not installed on a desktop machine. As a result locally compiled binaries may not be executable on the remote machine. In addition, some operating systems do not even offer pre-installed compilers and linkers.

A possible solution would be to set up a complex cross compiling environment. However the grid itself is heterogeneous, therefore this environment would have to be reconfigured in case the developer wants to run his application on a node with a different set-up. Another possibility is to transfer the sources to the remote machine, compile them there and finally transfer the executable back to the local machine where it is needed for debugging. These steps which have to be repeated each time the source code is changed, may get extremely annoying, especially in case of debugging where only small changes in the code are performed.

Therefore we added our own builder, the so called "Remote Builder" which can be activated for C/C++ Development Tooling (CDT)[4] Makefile projects. The remote builder uses GridFTP [1] for data transfer and glogin [13] for command invocation,

[4] http://www.eclipse.org/cdt/

both connections stay alive to improve responsiveness. The output of the build process is used to mark potential errors in the local source code editor.

5 Grid Application Launchers

After building a project, so-called launchers take care of running and debugging the applications. Usually in Eclipse, these launchers start the applications locally, but in a grid environment they should be started remotely on the grid resources.

For this reason we created our own launchers, in particular a newly developed launcher called "C/C++ Grid MPI Application Launcher". Again glogin is used to create interactive connections to the desired sites in the grid. Using this connection the application, which already resides on the remote host because of our remote builder, is started. In case of debugging, for each process of the parallel application, a GNU Project Debugger (GDB) [15] instance is started and attached to the respective MPI process. The standard input and output of those debuggers are redirected to network sockets, which in turn get forwarded to the local machine, where g-Eclipse is running, using glogin's traffic forwarding capabilities [14]. On the local side, instead of creating real instances of GDB, the forwarded streams of the remote debuggers are connected to the debugging functionality provided by the CDT. By doing so the debug perspective of Eclipse acts as if it were using a local gdb to debug the application. This circumvents the problems that would arise when using the combination of gdb and gdbserver for remote debugging, like having a local gdb matching the remote architecture and the need to have the shared libraries of the remote machine available locally. Due to the fact that we use the input and output of the remote debuggers, there is no need for a local GDB at all.

The most important fact is, that instead of allowing to debug only one process, we add every process of the MPI application to the application launch, thus allowing to conveniently control the whole remote running MPI application from our local machine. In addition to the launcher allowing to run and debug MPI applications on the grid, two further launchers were created. One allows to run and debug normal C/C++ applications on the grid, while the other one is used for running and debugging JAVA applications on the grid.

6 Trace Viewer

6.1 Visualization of Message Passing Programs

The Trace Viewer is a tool to visualise and analyse the communication of parallel message passing programs. As mentioned in the related work section above, several similar tools already exist. However many of these tools were developed some time

ago, thus relying on outdated GUI libraries, and some of them are only commercially available. This made it virtually impossible to integrate one of these tools into the Eclipse IDE. The integration, however, was one of our main objectives, in order to provide one workspace incorporating all tools.

Moreover few of these trace visualisation tools were designed to be extensible, especially not by using such a powerful concept like Eclipse's OSGi[5] based plug-in architecture. This extensibility however was needed in order to add the desired functionality and to allow to improve the Trace Viewers capabilities in the future.

Therefore a new component with attention to flexibility and extensibility is provided in g-Eclipse. This newly-created tool named Trace Viewer uses the Standard Widget Toolkit (SWT)[6] to draw the visualisations. The Trace Viewer is integrated into g-Eclipse but can be used as a stand-alone Rich Client Platform (RCP)[7] application as well. After careful consideration and with future fields of application in mind, four major points which need extensibility, were identified. To make use of one of the extension points a new plug-in must implement well defined interfaces which are provided within the basic Trace Viewer plug-in.

6.2 Trace Providers

The first extension point allows plug-ins to add trace providers which, as their name suggests, provide trace data. This trace data in turn can be displayed using plug-ins implementing the visualisation extension point.

An implementation of a trace provider must at least support logical clocks. Additionally it may also support lamport as well as physical clocks, provided that the data source offers such information.

A trace provides processes which can be queried by their process id. The processes allow, depending on the implementation, to query events according to their logical, lamport or physical clocks. An event can be of one of the four basic supported types namely: "send", "receive", "test" and "other". It is possible to provide information beyond the provided interfaces, that gets displayed in the properties view (Fig. 2) or can be used for trace format specific plug-ins.

Currently trace readers for two different trace formats are implemented. Both of these readers provide events with logical, physical and lamport clock information. The first reader allows to open the NOPE [6] trace format and is already in a mature state. The latter one adds preliminary support for the Open Trace Format (OTF) [8].

[5] http://www.osgi.org/

[6] http://www.eclipse.org/swt/

[7] http://www.eclipse.org/rcp/

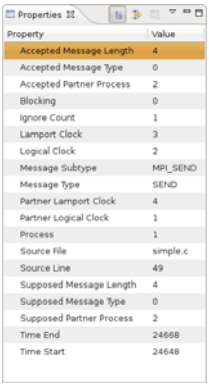


Fig. 2 Properties view showing detailed information about an selected event

6.3 Visualisations

The visualisation extension point allows visualisations to be added to the Trace Viewer. As already mentioned visualisations take care of displaying information made available by a trace provider. There are already three implementations which make use of this extension point. The first one (Fig. 3) visualises the trace data ac-

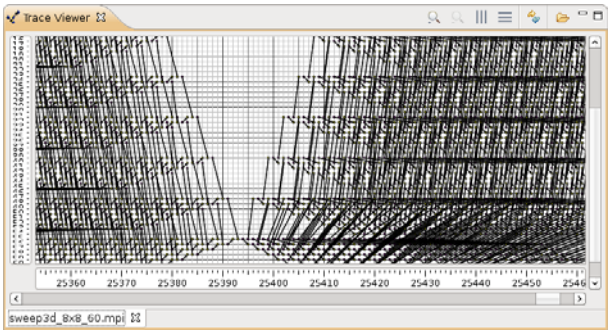


Fig. 3 Trace viewer showing a trace using lamport clocks

cording to the events' lamport clocks. The second visualisation (Fig. 4) uses the physical timestamps of the events. The last visualisation called "statistics visualisation" (Fig. 5) is just a proof of concept, which was developed to demonstrate the flexibility of this tool. It makes use of the Business Intelligence and Reporting Tools (BIRT)[8], which is also an Eclipse project, and its Chart Engine. Currently this visualisation just displays some statistical information, like the relation between the time spent with communication and the time spent doing actual calculation.

[8] http://www.eclipse.org/birt/

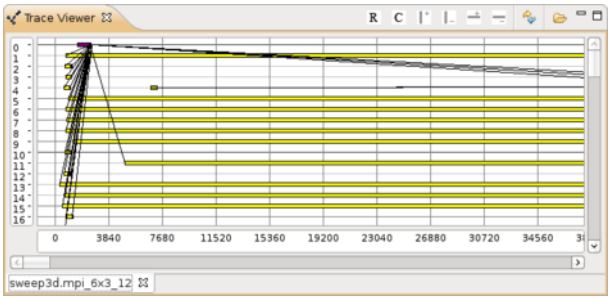


Fig. 4 Trace viewer showing a trace using physical clocks

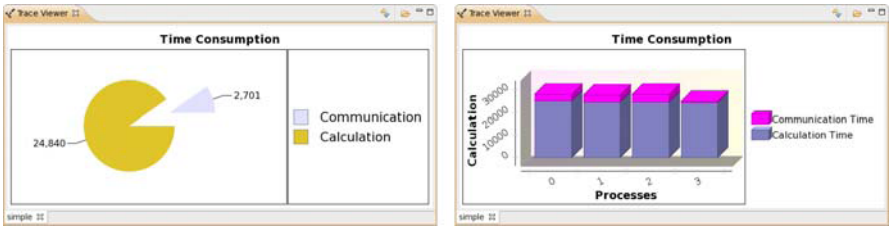


Fig. 5 Example visualisation plug-in showing statistical information about the trace

This can be done for all processes cumulatively using a pie chart, or for each process individually using a stacked bar chart, where each process is represented by a separate bar.

6.4 Actions

The actions extension point allows to register entries to a popup menu. This menu is displayed on right click on an event in the trace viewer and allows to perform actions on the selected event. It can be very useful to connect the trace viewer to other Eclipse components. Current implementations include a goto-source action and a breakpoint action. Both of these actions make use of the source file and source line information contained in events, provided for example by the NOPE trace file format. The goto-source action is relatively simple, on activation it searches the workspace for the appropriate source file, opens a source editor and jumps to the respective source line. The breakpoint action allows to visually set breakpoints on events in the graph. Since the source level debugger doesn't know the concept of events, the breakpoint action is more complex, because it has to deal with loops in the source code leading to multiple events per source line.

6.5 Markers

Another extension point allows to register so called markers. A marker can be used to alter the appearance of an event by changing its colour or shape. This functionality can be practical for different use cases. Tasks where it has already proven its viability are for example the *nope-*, *cause-effect-* and *debug-marker*. The NOPE Marker was developed, because the Trace Viewer only distinguishes between four different event types. However the NOPE trace format stores additional information about the specific MPI event type. This so called sub type information allows for example to distinguish between a `MPI_Send` and a `MPI_Isend` event. While without the NOPE Marker all events of one of the four types would look alike in the graphical representation, enabling it allows to give different sub types different appearances.

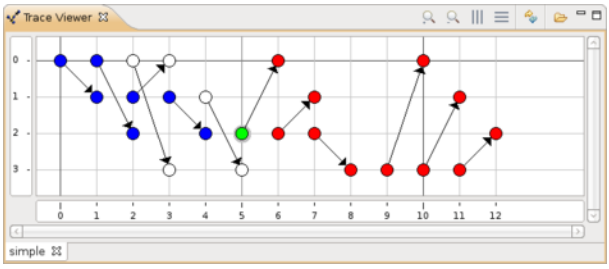


Fig. 6 Marker showing the cause-effect relationships of the events

The "cause effect" marker (Fig. 6) shows the relationships between a selected event and the other events in the trace by comparing their vector clocks. It marks events that affect the selected event, events that get affected by the selected event and independent events with different colours.

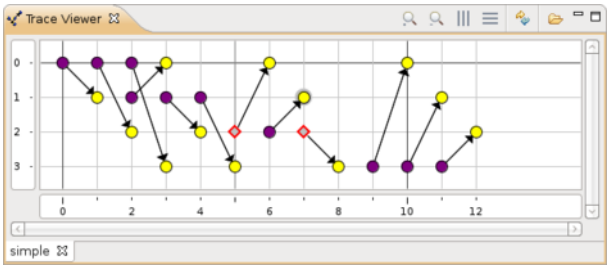


Fig. 7 Marker showing breakpoints on the events

The third provided marker is the breakpoint marker (Fig. 7). This marker allows to display the breakpoints created using the breakpoint action in the graph.

7 Conclusions and Future Work

The current version of g-Eclipse is already used in a series of grid projects. At this point in time, we are gathering experiences from the user to improve the functionality even further. For example, to provide an insight into the communication structures, especially of large traces, we are working on different pattern matching techniques. We introduce an algorithm to extract repeating communication patterns from MPI traces automatically to provide an easy and high-level understanding of the parallel application's communication behaviour. This would not only provide a high-level, abstract understanding of the behaviour of parallel applications, but would also support more directed performance optimization [12]. In addition, patterns also provide the possibility to quickly spot errors in the communication structure of an application, for example by revealing breaks in pattern sequences (Fig. 8) or by showing a mismatch between the actually recorded patterns and an application's intended structure. The main challenge is to efficiently detect such patterns from MPI event traces of long running and/or large scale applications. We implement the pattern search as a two step process: first we find locally repeating sequences on each process using a suffix tree algorithm and then match these local repeats with other sequences on other processes to generate global communication patterns.

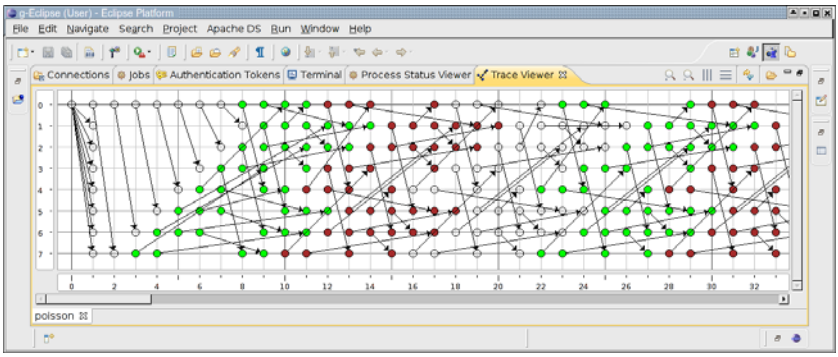


Fig. 8 Pattern matching plug-in that searches and marks repeating communication structures

References

1. Allcock, W.: GridFTP: Protocol Extensions to FTP for the Grid. Grid Final Document 20 (2003)
2. Friese, T., Smith, M., Freisleben, B.: GDT: A Toolkit for Grid Service Development. In Proc. of the 3rd International Conference on Grid Service Engineering and Management, 131–148 (2006)

3. Grabner, S., Kranzlmüller D., Volkert, J.: Debugging parallel programs using ATEMPT. In HPCN Europe 95: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, 235–240, Springer, London (1995)
4. Hood, R., Jost, G.: Debugger for Computational Grid Applications. HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop, 262, Washington DC (2000)
5. Kranzlmüller, D., Scarpa, M., Volkert, J.: DeWiz - A Modular tool Architecture for Parallel Program Analysis. In Euro-Par, 74–80 (2003)
6. Kranzlmüller, D., Volkert, J.: Nope: A Nondeterministic Program Evaluator. In ParNum 99: Proceedings of the 4th International ACPC Conference, 490–499, Springer, London (1999)
7. Kurniawan, D., Abramson, D.: Worqbench: An Integrated Framework for e-Science Application Development. In E-SCIENCE 06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing, 51, IEEE Computer Society, Washington (2006)
8. Malony, A.D., Nagel, W.E.: The open trace format (OTF) and open tracing for HPC. In SC 06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, 24, ACM, New York (2006)
9. Nagel, E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer, 12(1), 69–80, (1996)
10. Neophytou, N., Evripidou, P.: Net-dbx: A Java Powered Tool for Interactive Debugging of MPI Programs Across the Internet. In European Conference on Parallel Processing, 181–189 (1998)
11. Neophytou, P., Neophytou, N., Evripidou, P.: Net-dbx-G: a Web-based debugger of MPI programs over Grid environments. In CCGRID 04: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid, 35–42, IEEE Computer Society, Washington (2004)
12. Preissl, R., Schulz, M., Kranzlmüller, D., Supinski, B.R., Quinlan, D.J.: Using MPI Communication Patterns to Guide Source Code Transformations, Tools for Program Development and Analysis in Computational Science (2008)
13. Rosmanith, H., Kranzlmuller, D.: glogin - A Multifunctional, Interactive Tunnel into the Grid. (2004) doi: 10.1109/GRID.2004.33
14. Rosmanith, H., Volkert, J.: Traffic Forwarding with GSH/GLOGIN. In PDP 05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP05), 213–219, IEEE Computer Society, Washington (2005)
15. Stallman, R., Pesch, R., Shebs, S., et al.: Debugging with GDB. Free Software Foundation, 9th edition (2006) isbn: 1882114884
16. Wolniewicz, P., Meyer, N., Stroinski, M., Stuempert, M., Kornmayer, H., Polak, M., Gjermundrod, H.: Accessing Grid computing resources with g-Eclipse platform. In Computational Methods in Science and Technology, number 2 (2007)
17. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward Scalable Performance Visualization with Jumpshot. High Performance Computing Applications, 13(2), 277–288 (1999)

II
Parallel Communication and Debugging

Enhanced Memory debugging of MPI-parallel Applications in Open MPI

Shiqing Fan, Rainer Keller, and Michael Resch

Abstract In this paper, we describe the implementation of memory checking functionality based on instrumentation using `Valgrind-Memcheck` tool. The combination of `Valgrind` based checking functions within the MPI-implementation offers superior debugging functionalities, for errors that otherwise are not possible to detect with comparable MPI-debugging tools. The functionality is integrated into Open MPI as the so-called `memchecker`-framework. This allows other memory debuggers that offer a similar API to be integrated. The tight control of the user's memory passed to Open MPI, allows not only to find application errors, but also helps track bugs within Open MPI itself. We describe the actual checks, classes of errors being found, how memory buffers internally are being handled, show errors actually found in user's code and the performance implications of this instrumentation.

1 Introduction

Parallel programming with the distributed memory paradigm using the Message Passing Interface MPI [4] is often considered as an error-prone process. Great effort has been put into parallelizing libraries and applications using MPI. However when it comes to maintaining the software, optimizing for new hardware or even porting the code to other platforms and other MPI implementations, the developers will face additional difficulties [1]. They may experience errors due to hard-to-track timing critical bugs, deadlocks due to communication characteristics, MPI-implementation defined or even hardware dependent behavior. One class of bugs, that are hard-to-track are memory errors, specifically in non-blocking and one-sided communication.

Höchstleistungsrechenzentrum Stuttgart (HLRS), Nobelstrasse 19, 70550 Stuttgart, Germany, e-mail: {fan,keller,resch}@hlrs.de

In this paper, we introduce a debugging feature based on instrumentation functionalities offered by Memcheck [6] tool of Valgrind-tool suite [6], that is being employed within the Open MPI-library. The user's parameters, as well as other non-conforming MPI-usage and hard-to-track errors, such as accessing buffers of active non-blocking operations are being checked and reported. This kind of functionalities would otherwise not be detectable within traditional MPI-debuggers based on the PMPI-interface.

The structure of this paper is as follows: section 2 shows the basic idea and functionalities of Memcheck; section 3 gives an introduction to the design and implementation in both non-blocking and one-sided communication of Open MPI; in section 4, we show the performance implications of these two scenario; then in section 5 we present the real work, i.e. the errors, that are being detected and have been detected so far; finally, in section 6, we make a comparison with other available tools and concludes the paper with an outlook of the future work.

2 Overview of Memcheck

The tool suite Valgrind [6] may be employed on static and dynamic binary executables on x86/x86-64 and /PowerPC32/64-compatible architectures. It operates by intercepting the execution of the application on the binary level and interprets the instructions. With this instrumentation, Valgrind tools then may deduce information, and perform checks of different methodologies.

The system core of Valgrind provides a synthetic CPU. When the application starts, Valgrind will "trap" the real CPU, and run the machine code on its synthetic CPU, meanwhile, the debugging information is read from the executable and associated libraries. This instrumentation for the Valgrind-parser uses processor instructions that do not otherwise change the semantics of the application. By this special instruction preamble, Valgrind detects commands to steer the instrumentation. On the x86-architecture, the right-rotation instruction ror is used to rotate the 32-bit register edi, by 3, 13, 29 and 19, aka 64-Bits, leaving the same value in edi; the actual command to be executed is then encoded with an register-exchange instruction (xchgl) that replaces a register with itself (in this case ebx):

```
#define __SPECIAL_INSTRUCTION_PREAMBLE                \
             "roll $3,  %%edi ; roll $13, %%edi\n\t" \
             "roll $29, %%edi ; roll $19, %%edi\n\t" \
             "xchgl %%ebx, %%ebx\n\t"
```

Memcheck, a heavyweight memory checker in the Valgrind-tool suite, is well known for its tracking of memory definedness down to the bit level, which guarantees the partial defined bytes are also correctly dealt with. It stores two kinds of shadow memory values, for addressability and definedness, Memcheck shadows each byte in memory with the information presenting that whether the byte has been allocated (so-called A-Bits) and for each bit of the byte, whether it contains a

defined value (so-called V-Bits). With this AV-bit pair implementation, `Memcheck` is able to provide bit-precision checks of program errors as they run. It tracks the addressability of every byte of memory and the definedness of every bit of data in registers and memory, so that it can detect accesses to unaddressable memory errors and use of undefined value errors, such as buffer overruns and faulty access to stack. In total, every byte of memory is shadowed with 9 bits values (one A bit plus eight V bits). Memcheck also tracks all heap blocks allocated with `malloc()`, `new` and `new[]` to detect bad or repeated frees of heap blocks and memory leaks. Arguments to functions like `strcpy()` and `memcpy()`, are also checked for overlaps.

However, the disadvantage of using Memcheck is the slowdown of running applications, which is caused by adding code to check every memory access and every value computed. The size of the code is increased at least 12 times normally, and it runs 25-50 times slower than natively.

In this paper, we will describe the implementation of integrating `Memcheck` as a component of Open MPI, which helps MPI application and Open MPI developers track the wrongly use of memory, such as reading or writing to buffers of active, non-blocking Recv-operations and writing to buffers of active, one-sided Get-operations, as well as checking definedness of Open MPI-internal data structures, such as requests, communicators and datatype information.

3 Design and Implementation

In order to find MPI-related hard-to-track bugs in the application (and within Open MPI for that matter), we have taken advantage of an instrumentation-API offered by `Memcheck`. To allow other kinds of memory-debuggers, such as `bcheck` or Totalview's memory debugging features [8], we have implemented the functionality as a module into Open MPI's Modular Component Architecture [10]. The module is therefore called `memchecker` and may be enabled with the configure-option `--enable-memchecker`.

This may detect memory access bugs, such as buffer overruns and more, but also by knowledge of the semantics of calls like `strncpy`. However, `Valgrind` does not have any knowledge of the semantics of MPI-calls. Also, due to the way, how `Valgrind` is working, errors due to undefined data may be reported late, way down in the call stack. The original source of error in the application therefore may not be obvious.

3.1 Non-blocking Communication

In Open MPI objects such as communicators, types and requests are declared as pointers to structures. These objects when passed to MPI-calls are being immediately checked for definedness and together with `MPI_Status` are checked upon

exit[1]. Memory being passed to Send-operations is being checked for accessibility and definedness, while pointers in Recv-operations are checked for accessibility, only.

Reading or writing to buffers of active, non-blocking Recv-operations and writing to buffers of active, non-blocking Send-operations are obvious bugs. Buffers being passed to non-blocking operations (after the above checking) is being set to undefined within the MPI-layer of Open MPI until the corresponding completion operation is issued. This setting of the visibility is being set independent of non-blocking `MPI_Isend` or `MPI_Irecv` function. When the application touches the corresponding part in memory before the completion with `MPI_Wait`, `MPI_Test` or multiple completion calls, an error message will be issued. In order to allow the lower-level MPI-functionality to send the user-buffer as fragment, the lower-layer BTLs (Byte Transfer Layers) are adapted to set the fragment in question to accessible and defined, as may be seen in Fig. 1. Care has been taken to handle derived datatypes and it's implications.

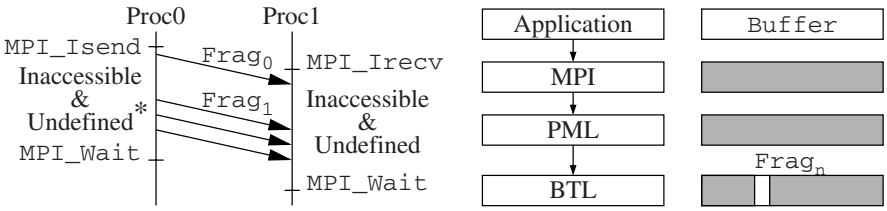


Fig. 1 Fragment handling to set accessibility and definedness, non-blocking communication

For Send-operations, the MPI-1 standard also defines, that the application may not access the send-buffer at all (see [4], p. 30). Many applications do not obey this strict policy, domain-decomposition based applications that communicate ghost-cells, still read from the send-buffer. To the authors' knowledge, no existing implementation requires this policy, therefore the setting to undefined on the Send-side is only done when strict-checking is enabled (see Undefined* in Fig. 1).

3.2 One-sided Communication

For one-sided communications, MPI-2 standard defines that, any conflicting accesses to the same memory location in a window are erroneous (see [5], p. 112). If a location is updated by a put or a accumulate operation, then this location cannot be accessed by a load or another RMA operation until the updating operation is completed on the target. If a location is fetched by a get operation, this location

[1] E. g. this showed up uninitialized data in derived objects, e. g. communicators created using `MPI_Comm_dup`

cannot be accessed by other operations as well. When a synchronization call starts, the local communication buffer of an RMA call and a get call should not be updated until it is finished. User buffer of `MPI_Put` or `MPI_Accumulate`, for instance, are set not accessible when these operations are initiated, until the completion operation finished (see Fig. 2). `Valgrind` will produce an error message, if there is any read or write to the memory area of the user buffer before corresponding completion operation terminates.

In Open MPI, there are two One-sided communication modules, point-to-point and RDMA. Similar checks has been implemented for `MPI_Get`, `MPI_Put`, `MPI_Fence` and `MPI_Accumulate` in point-to-point module.

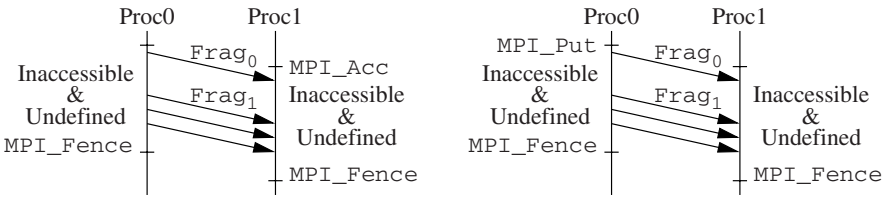


Fig. 2 Fragment handling to set accessibility and definedness, one-sided communication

4 Performance Implications

Adding instrumentation to the code does induce a slight performance hit due to the assembler instructions as explained above, even when the application is not run under `Valgrind`.

Tests have been done for both non-blocking communication and one-sided communication with several benchmarks, all of which were run on the `DGrid`-cluster at HLRS. This machine consists of dual-processor Intel Woodcrest, using Infiniband-DDR network with the OpenFabrics stack.

4.1 Non-blocking communication performance

For IMB, two nodes were used to test in following cases: compilation with&without `--enable-memchecker` and with `--enable-memchecker` but disabled MPI-object checking (see Fig. 3) and with&without `Valgrind` was run (see Fig. 4). We include the performance results on two nodes using the PingPong test. In Fig. 3 the measured latencies (left) and bandwidth (right) using Infiniband (not running with `Valgrind`) shows the costs incurred by the additional instrumentation, ranging from 18 to 25% when the MPI-object checking is enabled as well, and 3-6% when memchecker is enabled, but no MPI-object checking is performed. As one may note, while latency is sensitive to the instrumentation added, for larger packet-

sizes, it is hardly noticeable anymore (less than 1% overhead). Figure 4 shows the cost when additionally running with `Valgrind`, again without further instrumentation compared with our additional instrumentation applied, here using TCP connections employing the IPoverIB-interface.

The large slowdown of the MPI-object checking is due to the tests of every argument and its components, i. e. the internal data structures of an `MPI_Comm` consist of checking the definedness of 58 components, checking an `MPI_Request` involves 24 components, while checking `MPI_Datatype` depends on the number of the base types.

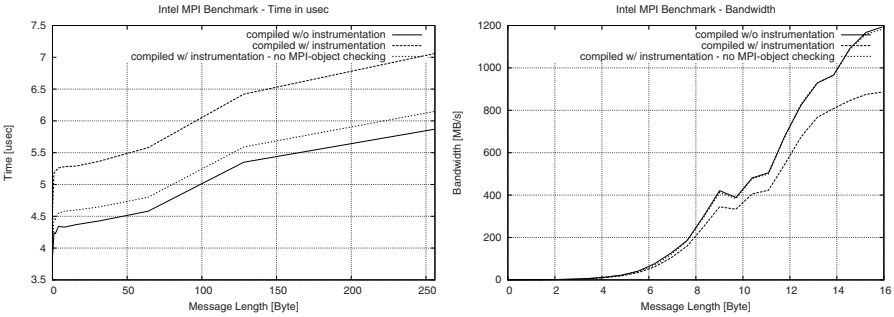


Fig. 3 Latencies and bandwidth with&without memchecker-instrumentation over IB, running without `valgrind`

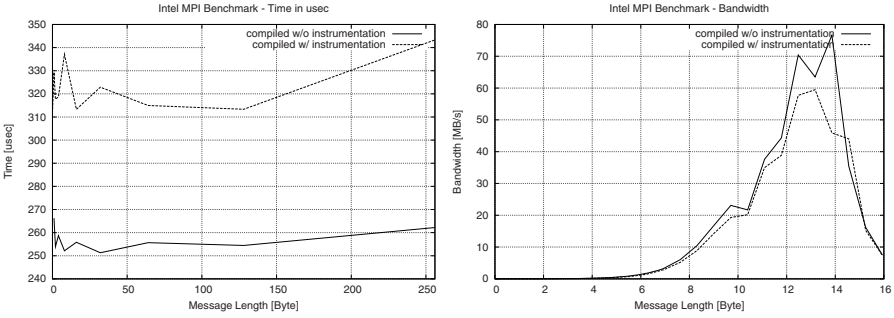


Fig. 4 Latencies and bandwidth with&without memchecker-instrumentation using IPoverIB, running with `valgrind`

The BT-Benchmark has several classes, which have different complexity, and data size. The algorithm of BT-Benchmark solves three sets of uncoupled systems of equations, first in the x, then in the y, and finally in the z direction. The tests are done with sizes Class A and Class B. Figure 5 shows the time in seconds for the BT Benchmark. The Class A (size of 64x64x64) and Class B (size of 102x102x102) test was run with the standard parameters (200 iterations, time-step *dt* of 0.0008).

Again, we tested Open MPI in the following three cases: Open MPI without memchecker component, running under `Valgrind` with the memchecker component disabled and finally with `--enable-memchecker`.

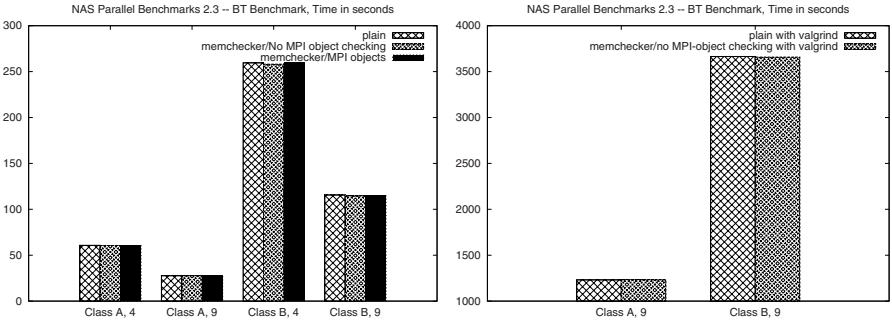


Fig. 5 Time of the NPB/BT benchmark for different classes running without (left) and with (right) `valgrind`

As may be seen and is expected this benchmark does not show any performance implications whether the instrumentation is added or not. Of course due to the large memory requirements, the execution shows the expected slow-down when running under `Valgrind`, as every memory access is being checked.

4.2 One-sided communication performance

For one-sided communication, we used NetPIPE and Intel MPI Benchmark both on two nodes of `DGrid`-cluster at HLRS, and MPI-object checking is disabled for all tests in this case, as it will result the same large slowdown as we explained in section 4.1.

In IMB benchmark bi-directional put and get are used, both in aggregate mode, i.e. both tests will run with varying transfer sizes in bytes which is issued by the corresponding one sided communication call, and timings will be averaged over multiple samples. The bi-directional benchmarks are exact equivalents of the message passing PingPing. All tests were run in following cases, with/without memchecker implementation, and run with/without `Valgrind`.

Figure 6 presents the average time of running bi-directional get and put tests with and without the memchecker implementation running without `Valgrind`. The performance of `MPI_Get` (see left side of Fig. 6) in these cases is nearly identical, and the one with memchecker implementation is losing only 1% of run time. For `MPI_Put` (see right side of Fig. 6), we got similar result as `MPI_Get`. However, notably, `MPI_Put` has a better performance than `MPI_Get` in general. There are several factors affecting the performance of `MPI_Put` transfer, for example the choice of window location and the shape and location of the origin and target buffer. Transfers to a target window in memory allocated by `MPI_ALLOC_MEM` may be

much faster on shared memory systems; transfers from contiguous buffers will be faster on most systems; the alignment of the communication buffers may also impact performance, see [5] p. 114.

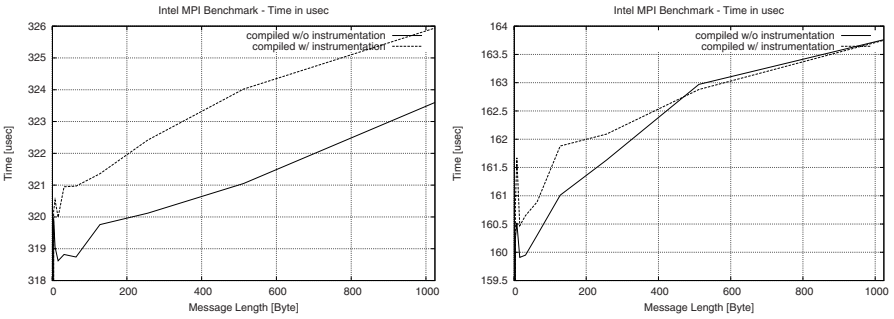


Fig. 6 Time for bi-directional get(left) and put(right), running without `valgrind`

On the other hand, the performance is dropping down a lot when running with `Valgrind`, as shown in Fig. 7, the results of the same test but running with `Valgrind`. In this case, the additional cost of the memchecker instrumentation (less than 1% of run time) is almost negligible.

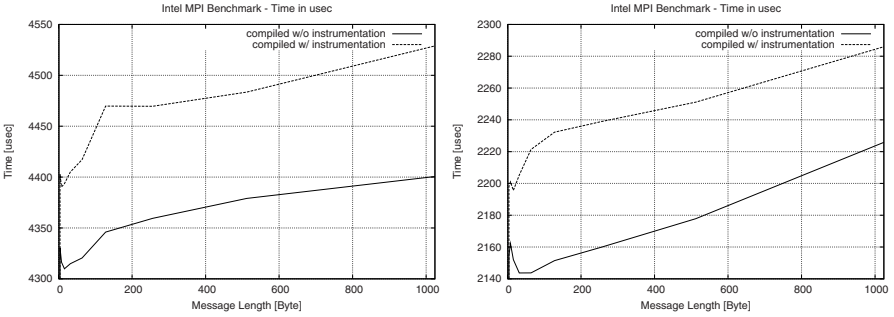


Fig. 7 Time for bi-directional get(left) and put(right), running with `valgrind`

NetPIPE is a protocol independent performance tool that presents the network performance under a variety of conditions. It performs PingPong tests between two processes with increasing message size through different protocols and MPI implementations. The message sizes are chosen at regular intervals with slight perturbations. Each data point involves many ping-pong tests to get a accurate timing value. Here it was modified for testing the performance of Open MPI. Namely the variables of window and address pointers were adapted, all of which are not performance relevant.

The performance of `MPI_Get` and `MPI_Put` is shown in Fig. 8 and Fig. 9 separately, running without `Valgrind`, each of the figures presents the run time

and bandwidth of executing the application. As seen from the figures, the application got 3%-5% performance loss, when memchecker is enabled.

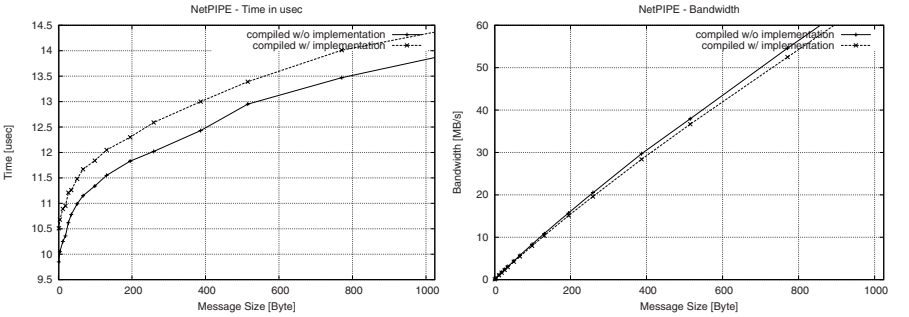


Fig. 8 Time and bandwidth, one-sided get, running without `valgrind`

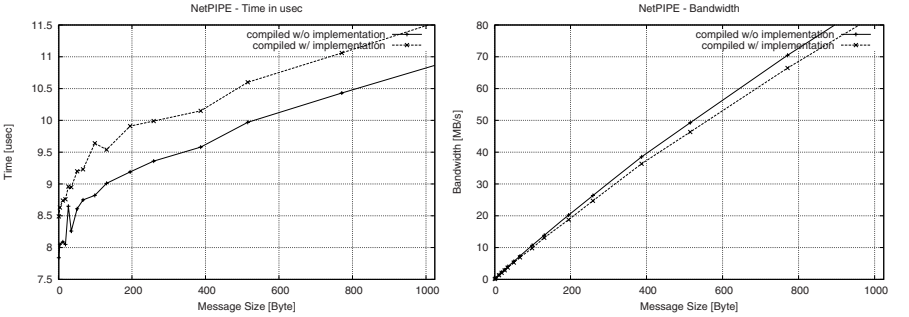


Fig. 9 Time and bandwidth, one-sided put, running without `valgrind`

5 Detectable error classes and findings in actual applications

The kind of errors, detectable with a memory debugging tool such as `Valgrind` in conjunction with instrumentation of the MPI-implementation are:

- Wrong input parameters, e. g. wrongly sized send buffers:

```
char * send_buffer;
send_buffer = malloc (5);
memset(send_buffer, 0, 5);
MPI_Send(send_buffer, 10, MPI_CHAR, 1, 0, \
         MPI_COMM_WORLD);
```

- Uninitialized input buffers:

```
char * buffer;
buffer = malloc (10);
MPI_Send(buffer, 10, MPI_INT, 1, 0, \
         MPI_COMM_WORLD);
```

- Usage of the uninitialized MPI_ERROR-field of MPI_Status[2]:

```
MPI_Wait(&request, &status);
if(status.MPI_ERROR != MPI_SUCCESS)
    return ERROR;
```

- Writing into the buffer of active non-blocking Send or Recv-operation or persistent communication:

```
int buf = 0;
MPI_Request req;
MPI_Status status;
MPI_Irecv(&buf, 1, MPI_INT, 1, 0, \
          MPI_COMM_WORLD, &req);
/* Will produce a warning */
buf = 4711;
MPI_Wait(&req, &status);
```

- Read from the buffer of active non-blocking Send-operation in strict-mode:

```
int inner_value = 0, shadow = 0;
MPI_Request req;
MPI_Status status;
MPI_Isend(&shadow, 1, MPI_INT, 1, 0, \
          MPI_COMM_WORLD, &req);
/* Will produce a warning */
inner_value += shadow;
MPI_Wait(&req, &status);
```

- Read from the buffer of active accumulate operation:

```
MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, \
               MPI_COMM_WORLD, &win);
MPI_Win_fence (0, win);
MPI_Accumulate(A, NROWS*NCOLS, MPI_INT, 1, 0, 1, \
               xpose, MPI_SUM, win);
```

[2] The MPI-1 standard declares the MPI_ERROR-field to be undefined for single-completion calls such as MPI_Wait or MPI_Test (p. 22).

```
/* Will produce a warning */
printf("\n%d\n",A[0][0]);
MPI_Win_fence(0, win);
```

- Write to the buffer of active get operation:

```
MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, \
               MPI_COMM_WORLD, &win);
MPI_Win_fence(0, win);
MPI_Get(A, NROWS*NCOLS, MPI_INT, 1, 0, 1, \
        xpose, win);
/* Will produce a warning */
A[1][0] = 4711;
MPI_Win_fence(0, win);
```

- Uninitialized values, e. g. MPI-objects from within Open MPI.

During the course of development, several software packages have been tested with the memchecker functionality. Among them problems showed up in Open MPI itself (failed in initialization of fields of the status copied to user-space), an MPI testsuite [2], where tests for the `MPI_ERROR` triggered an error. In order to reduce the number of false positives Infiniband-networks, the `ibverbs`-library of the OFED-stack [7] was extended with instrumentation for buffer passed back from kernel-space.

6 Conclusion and future work

We have presented an implementation of memory debugging features into Open MPI, using the instrumentation of the `Valgrind`-suite, and the performance implication of using the instrumentation with several benchmarks. This allows detection of hard-to-find bugs in MPI-parallel applications, libraries and Open MPI itself [1]. This is new work, up to now, no other debugger is able to find these kind of errors.

The future work will be mainly focused on capturing and restoring the memory states in Open MPI, i.e. capturing and restoring the AV-bit pairs from `Valgrind`. This is necessary for setting the accessibility of the user buffer more precisely and for preventing overwriting the states of the memory location. For instance, a snapshot of the user buffer states is captured and stored when non-blocking operation starts, then the buffer set to be not accessible, which will allow `Valgrind` to detect the memory access. When send operation is finished, the snapshot will be restored back to the corresponding memory location, so that the states of the user buffer remains unchanged.

With regard to related work, debuggers such as Umpire [9], Marmot [3] or the Intel Trace Analyzer and Collector [1], actually any other debugger based on the

Profiling Interface of MPI, may detect bugs regarding non-standard access to buffers used in active, non-blocking communication without hiding false positives of the MPI-library itself.

Acknowledgements This work was funded by project Int.EU.Grid (Interactive EUropean Grid) with EU-Contract Number 031857, by project DORII (Deployment of Remote Instrumentation Infrastructure) with EU-Contract Number 213110 and by Microsoft Technical Computing Initiative (TCI) since March 2007.

We would like to thank Julian Seward and the open source community for `Valgrind`, which has proven invaluable in many software projects.

References

1. DeSouza, J., Kuhn, B., de Supinski, B.R.: Automated, scalable debugging of MPI programs with Intel message checker. In: Proceedings of the 2nd International Workshop on Software engineering for high performance computing system applications, vol. 4, pp. 78–82. ACM Press, NY, USA (2005)
2. Keller, R., Resch, M.: Testing the correctness of MPI implementations. In: Proceedings of the 5th Int. Symp. on Parallel and Distributed Computing conference, pp. 291 – 295. Timisoara, Romania (2006)
3. Krammer, B., Müller, M.S., Resch, M.M.: Runtime checking of MPI applications with Marmot. In: PARCO'05. Malaga, Spain (2005)
4. Message Passing Interface Forum: MPI: A Message Passing Interface Standard (1995). `http://www.mpi-forum.org`
5. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface (1997). `http://www.mpi-forum.org`
6. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the USENIX'05 Annual Technical Conference. Anaheim, CA, USA (2005)
7. The Open Fabrics project webpage. WWW (2007). `https://www.openfabrics.org`
8. Totalview Memory Debugging capabilities. WWW. `http://www.etnus.com/TotalView/Memory.html`
9. Vetter, J.S., de Supinski, B.R.: Dynamic software testing of MPI applications with Umpire. In: Proceedings of Supercomputing (SC) (2000). `http://www.sc2000.org/proceedings/techpapr/index.htm`
10. Woodall, T., Graham, R., Castain, R., Daniel, D., Sukalski, M., Fagg, G., Gabriel, E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A.: Open MPI's TEG Point-to-Point Communications Methodology: Comparison to Existing Implementations. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, vol. 3241, pp. 105–111. Springer, Budapest, Hungary (2004)

MPI Correctness Checking with Marmot

Bettina Krammer, Tobias Hilbrich, Valentin Himmler, Blasius Czink, Kiril Dichev, and Matthias S. Müller

Abstract Parallel programming is a complex, and since the multi-core era has dawned, also a more and more common task that can be alleviated considerably by tools supporting the application development and porting process. The Message Passing Interface (MPI) is widely used to write parallel programs using message passing, but it does not guarantee portability between different MPI implementations. When an application runs without any problems on one platform but crashes or gives wrong results on another platform, developers tend to blame the compiler/architecture/MPI implementation. In many cases the problem is a subtle programming error in the application undetected on the platforms used previously. Finding this bug can be a very strenuous and difficult task. In this paper we present the Marmot tool, an automated correctness checker for MPI applications during runtime. Examples of violations of the MPI standard are the introduction of irreproducibility, deadlocks, incorrect management of resources such as communicators, groups, datatypes etc. or the use of non-portable constructs. To cover different aspects of correctness debugging in a user-friendly environment, also in hybrid applications using both MPI and OpenMP, we also work on coupling Marmot with a parallel debugger (DDT) or a threading tool (Intel® Thread Checker). Some examples of experiences with real-world applications are given.

Bettina Krammer, Valentin Himmler, Blasius Czink, Kiril Dichev
High Performance Computing Center Stuttgart (HLRS), Universität Stuttgart, Nobelstrasse 19, 70569 Stuttgart, Germany, e-mail: {krammer,himmler,czink,dichev}@hlrs.de

Tobias Hilbrich, Matthias S. Müller
TU Dresden, Center for Information Services and High Performance Computing (ZIH), 01062 Dresden, Germany, e-mail: {Tobias.Hilbrich@zih.tu-dresden.de,matthias.mueller@tu-dresden.de}

1 Introduction

The Message Passing Interface (MPI) has been a commonly used standard [1, 2] for writing parallel programs for more than a decade, at least within the High Performance Computing (HPC) community. With the arrival of multi-core processors, parallel programming paradigms such as MPI or OpenMP will become more popular among a wider public in many application domains as software needs to be adapted and parallelised to exploit fully the processor's performance. However, tracking down a bug in a distributed program can turn into a very painful task, especially if one has to deal with a huge and hardly comprehensible piece of legacy code. The main difficulties are:

1. Developers do not only have to face all the problems that occur in serial programming. In addition, parallel applications get more and more complex and especially with the introduction of optimisations like the use of non-blocking communication also more error prone.
2. MPI programs do not always behave deterministically. Deadlocks or race conditions may appear, depending on the platform environment or on the MPI implementation. What is worse, they may only appear sometimes or only when running on a very high number of processes. Thus, it may take users or developers quite a long time until they even realise that the program gives wrong results, but only sometimes. Unfortunately, it may be impossible to reproduce these errors, and the errors may never occur in the presence of a debugging tool as any sort of surveillance slightly changes the program behaviour (so-called *Heisenbugs*).
3. The MPI standard leaves many decisions to the implementation, e.g. whether or not a standard communication is blocking or how to implement so-called *opaque* objects. This implementation-defined behaviour may cause problems when porting an application from one platform to another.

In the following sections, we first give a short overview on related work (Sect. 2) and describe the design of Marmot as well as possible checks for MPI and hybrid programs (Sect. 3) and collaboration with other tools (Sect. 4). Finally, we present some experiences with real-world applications (Sect. 5) and give a very short user-guide (Sect. 6) and some concluding remarks and an outlook to work planned in future (Sect. 7).

2 Related Work

Finding bugs in a complex parallel application is quite a painful task. Fortunately there are powerful tools for the different aspects of debugging, e.g. tools for memory checking or for correctness checking. Apart from the classical way of debugging – `printf` statements – the different solutions are roughly grouped into four different approaches: classical debuggers, special MPI libraries and other tools that may perform a runtime or post-mortem analysis.

1. The freely available debugger gdb [19], which is also used with its graphical front-end ddd [20], has currently no support for MPI, but it can be attached to one or several, possibly already running MPI processes. The same can be done with special memory-checking debuggers, e.g. Valgrind [22, 23]. More convenient are parallel debuggers, which are based on serial debuggers such as gdb. They provide the usual interactive functionality of debuggers, such as single-stepping, breakpointing, evaluating variables, etc., but additionally allow the user to monitor and act on groups of processes in a single debugging session. Examples are the well-known commercial debuggers Totalview [17] or DDT [16]. These debuggers can also be used for a post-mortem analysis of core files.
2. The second approach is to provide a special debug version of the MPI library (e.g. MPIch or NEC-MPI). This version is not only used to catch internal errors in the MPI library, but also to detect some incorrect usage of MPI by the user, e.g. a type mismatch of sending and receiving messages or mismatched collective operations [4, 5, 6].
3. Another possibility is to develop tools dedicated to finding problems within MPI applications at runtime. At present, four known different message-checking tools are under more or less active development. MPI-CHECK [8] is currently restricted to Fortran code and performs argument type checking or finds problems like deadlocks [8]. Similar to Marmot [9, 10], Umpire [3] uses the profiling interface. The newest kid on the block is the MPI correctness checker library that is integrated in the Intel® Trace Analyzer and Collector [26]. It is based on the previous Intel® Message Checker (IMC) [25], which was at that time an example of the fourth approuch.
4. The fourth approach is to perform a post-mortem analysis by collecting all information on MPI calls in a trace file. After program execution, this trace file is analysed by a separate tool or compared with the results from previous runs [7]. This approach is also used by many tools with respect to performance analysis, and indeed, in some cases it can be very enlightening to "abuse" a performance tool for debugging.

As no tool is an all-in-one device suitable for every purpose, a combination of different tools will probably aid the developers most. While a memory-checking debugger may be able to diagnose that an application crashes due to an uninitialized variable, it will definitely not help much in finding incorrect usage of the MPI interface as Marmot does. Therefore, we also aim at collaborating with other tools, see Sect. 4. Regardless, not every error can be caught by tools.

3 Design of Marmot

Marmot [10, 11, 12] is a library that uses the so-called PMPI profiling interface to intercept MPI calls and analyse them during runtime. It has to be linked to the application in addition to the underlying MPI implementation, not requiring any modification of the application's source code nor of the MPI library. The tool checks

if the MPI API is used correctly and checks for errors frequently made in MPI applications, e.g. deadlocks, the correct construction and destruction of resources, etc. It also issues warnings for non-portable behaviour, e.g. using tags outside the range guaranteed by the MPI standard.

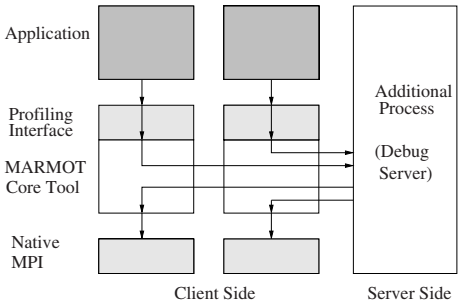


Fig. 1 Design of Marmot

Figure 1 illustrates the design of Marmot. Local checks including verification of arguments such as tags, communicators, ranks, etc. are performed on the client side. An additional MPI process (referred to as *debug server*) is added for the tasks that cannot be handled within the context of a single MPI process, e.g. deadlock detection. Another task of the debug server is the logging and the control of the execution flow. Every client has to register at the debug server, which gives its clients the permission for execution in a round-robin way. Information is transferred between the original MPI processes and the debug server using MPI. The disadvantage of this server/client architecture is that it inflicts a bottleneck, thus affecting the scalability and performance of the tool, especially for communication-intensive applications [12].

In order to ensure that the debug server process is transparent to the application, we map `MPI_COMM_WORLD` to a Marmot communicator that contains only the application processes. Since all other communicators are derived from `MPI_COMM_WORLD` they will also automatically exclude the debug server process. This mapping is done at start-up time in the `MPI_Init` call, where we also map all other predefined MPI resources, such as groups or datatypes, to our own Marmot resources. When an application constructs or destructs resources during run-time, e.g. by creating or freeing a user-defined communicator, the Marmot maps are updated accordingly. Having its own book-keeping of MPI resources, independently of the actual MPI implementation, Marmot can thus verify correct handling of resources.

The output of Marmot is available in different formats, e.g. as text log file or html/xml file, which can be displayed and analysed using a web browser or graphical interface. An excerpt from Marmot's HTML output is depicted in Fig. 2.

Marmot is intended to be a portable tool that has been tested on many different platforms and with many different MPI implementations, for instance Linux Clusters with IA32/IA64/EM64T processors, Cray, Hitachi, IBM Regatta and NEC SX

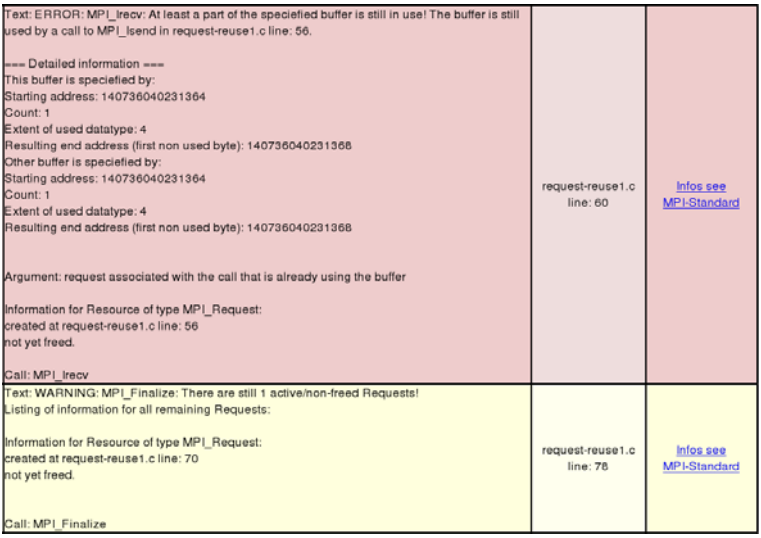


Fig. 2 Excerpt from Marmot's HTML output

systems, using different compilers (GNU, Intel, PGI,etc.) and different MPI implementations (MPIch, Open MPI, LAM/MPI, vendor MPIs, etc.). Functionality and performance tests are performed with test suites, microbenchmarks and real applications [11, 12].

Marmot supports the complete MPI-1.2 standard for C and Fortran applications and is being extended to also cover MPI-2 functionality.

3.1 Possible Checks for MPI Applications

Parallel programming is a complex challenge. It offers enough pitfalls that MPI can imaginably stand for "Maddening Programming Interface". Among the Top Ten common programming errors are:

- **Deadlocks:** Marmot contains a mechanism to automatically detect deadlocks and notify the user where and why they have occurred. In general, deadlocks are caused by the non-occurrence of something else, for example mismatched send/receive operations or mismatched collective calls. One can distinguish between *real* deadlocks, which occur inevitably, and *potential* deadlocks, which may occur only under certain circumstances, e.g. depending on data races or on the implementation, for instance, if a standard send is implemented as a buffered send or not. In this code snippet process 0 and process 1 exchange messages between each other.

```
if (rank == 0) {
```

```
    // send to 1 and receive from 1
    MPI_Send(...);
    MPI_Recv(...);
  } else if (rank == 1) {
    // send to 0 and receive from 0
    MPI_Send(...);
    MPI_Recv(...);
  }
```

If the MPI_Send is implemented in buffered mode, for example for small message sizes, this code will not deadlock, otherwise it will. Currently Marmot's deadlock detection is based on a timeout mechanism and therefore finds all real deadlocks. Marmot's debug server surveys the time each process is waiting in an MPI call. If this time exceeds a certain user-defined limit on all processes at the same time, the debug process issues a deadlock warning. The user is then able to trace the last few calls on each node. It is also possible that attaching Marmot (or any other tool) to an application slightly changes the execution flow in such a way that a potential deadlock becomes apparent.

- **Data races:** Potential race conditions can be caused by various reasons, e.g. by the use of a receive call with the wildcard MPI_ANY_SOURCE as source argument or the wildcard MPI_ANY_TAG as tag argument, by the use of random numbers, or by the fact that nodes do not behave exactly the same. Some users also rely on collective calls being synchronising, however, the only synchronising collective call is the MPI_Barrier. Other collective calls can be synchronising or not, depending on their implementation. For example, assume that any of the send calls on the processes 1 and 2 match to any of the receive calls on process 0.

```
if (rank == 0) {
  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,...);
  MPI_Bcast(....);
  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,...);
} else if (rank == 1) {
  MPI_Send(...);
  MPI_Bcast(...);
} else if (rank == 2){
  MPI_Bcast(...);
  MPI_Send(...);
}
```

If the MPI_Bcast is synchronising process 0 will have to receive the message from process 1 first. If it is not then the message order will not be deterministic: either the message from process 1 or from process 2 can be received first. At present, Marmot indicates the use of wildcards, but it does not construct dependency graphs to view the different possible executions nor does it use methods like record and replay to identify and track down bugs in parallel programs [7] or to compare different runs. Why does one need a tool to detect this sort of argument as a simple *grep* command on the source code would give the same result?

Actually, a search command does neither show the execution flow nor will it be able to detect this argument if the application takes functions from some other library with hidden MPI calls.

- **Mismatches:** Mismatches in arguments of one call can be detected locally and are sometimes even detected by the compiler. Examples are wrong type or number of arguments. Mismatches are also seen in arguments involving more than one call, e.g. in send/receive pairs or in collective calls. Special attention is needed when comparing matched pairs of derived datatypes because it is legal to send, for example two (`MPI_INT`, `MPI_DOUBLE`) and to receive one (`MPI_INT`, `MPI_DOUBLE`, `MPI_INT`, `MPI_DOUBLE`), or to send one (`MPI_INT`, `MPI_DOUBLE`) and to receive one (`MPI_INT`, `MPI_DOUBLE`, `MPI_INT`, `MPI_DOUBLE`) (a so-called *partial* receive). MPI implementations usually abort an application when there is a datatype mismatch, e.g. send an `MPI_INT` and receive an `MPI_DOUBLE`, but no exact diagnosis of the mismatch is given.
- **Resource handling:** This is an area in MPI where incorrect usage may result in fatal errors with almost no obvious link to the real cause. Since they are very difficult to find, we place special focus into detecting them. Marmot is able to keep track of the proper construction, usage and destruction of all MPI resources, such as communicators, groups, datatypes, etc. As these resources are "opaque" objects and therefore implementation-dependent, Marmot has its own book-keeping of these resources and, thus, duplicates the management done by the underlying MPI library. Marmot also checks if requests and other arguments (tags, ranks, etc.) are used correctly, e.g. if an active requests is reused. The main functionality is implemented for the C language binding, whereas the functionality for the Fortran language binding is obtained through a wrapper to the C interface. Special attention is paid to the verification of the datatypes because they are one of the major differences between the C and the Fortran language binding.
- **Memory and other resource exhaustion:** Non-blocking calls such as `MPI_Isend` etc. can complete without issuing a matching test or wait call. However, the number of available request handles is limited (and implementation defined). Therefore requests should always be freed, as should allocated communicators, datatypes, etc. Marmot gives a warning when a request is reused, and also when there are active or non-freed requests left at the `MPI_Finalize`. Another issue is reusing memory that is still in use, for example by reading/writing from/into a buffer by an unfinished send/receive operation. Marmot does currently not perform any checks if a buffer can be reused safely because the transmission of data has completed. This kind of check is a subtle task that requires some insight into an MPI implementation: what is really going on when calling e.g. `MPI_Issend` or `MPI_Irecv`, how does that depend on the message size, etc.? In some cases, Marmot checks if buffers are overwritten by mistake, e.g. for `MPI_Gatherv` and similar collective calls, it is verified if on the root process data is overridden due to an erroneous array of displacements.
- **Portability:** The MPI standard leaves many decisions to the implementors, for example how to implement opaque objects and handles to these objects, if to

implement `MPI_Send` as buffered call or not, if to implement collective calls as synchronising calls, if to make the implementation thread-safe or not, etc. Some of these issues can already be detected at compile time when the application is ported to another environment, some can be found at runtime by Marmot, e.g. using a tag beyond the guaranteed limit.

Marmot supports the complete MPI-1.2 standard, although not all possible tests (such as consistency checks) are implemented yet. It can be used with any standard-conforming MPI implementation and may thus be deployed on any development platform available to the programmer. Although high-quality MPI implementations detect some of these errors themselves, there are many cases where they do not give any warnings. For example, non-portable implementation-specific behaviour is not indicated by the implementation itself, nor are checks performed that would decrease the performance too much, such as consistency checks. What is worse, MPI implementations tolerate quite a few errors without warnings or crashing, by simply giving wrong results.

3.2 Possible Checks for Hybrid Applications

As HPC systems tend to use steadily increasing amounts of computing cores, it is necessary to provide strategies to utilize these systems. For some systems it is gainful to use MPI and multi-threading at the same time. So called "hybrid" applications follow this strategy and usually use one MPI process per computing node and one thread for each computing core of a node. This also increases the complexity of applications and introduces new MPI usage errors. In order to support development of such applications it is possible to use Marmot for hybrid OpenMP/MPI applications.

The existence of threads has consequences as it is possible to call the MPI in parallel, by using multiple threads. The MPI-2 standard restricts the multi threaded usage by introducing four different usage levels:

`MPI_THREAD_SINGLE:`	only one thread exists
`MPI_THREAD_FUNNELED:`	multiple threads may exist but only the main thread (i.e. the thread that initialized MPI) performs MPI calls
`MPI_THREAD_SERIALIZED:`	multiple threads exist and each thread may perform MPI calls as long as no other thread is calling MPI
`MPI_THREAD_MULTIPLE:`	multiple threads may call MPI simultaneously

These levels are referred to as "thread levels". Each MPI application may specify a required thread level, but an MPI implementation may return a lower level instead. In addition to the specification of the thread levels there are further restrictions in the MPI standard. An example of such a restriction is the usage of communicators in collective calls, namely communicators must not be used simultaneously in multiple collective calls of one process.

In order to use Marmot in hybrid applications it is necessary to synchronize it to avoid unprotected parallel data access within Marmot. This is done by using commands of the used multi threading paradigm. Currently only OpenMP is supported but implementations for other paradigms are feasible. So in addition to the normal checks for MPI usage it is necessary to check for the following usage faults:

- **Conformance to the provided thread level:** The thread level provided by the MPI implementation must not be violated. In order to check this it is necessary to observe which threads are calling MPI and whether it is possible that multiple threads call MPI simultaneously on one process. Marmot does this at runtime and detects violations to the provided thread level if they actually occur in a run. In addition, an application should require the lowest sufficient thread level. In order to aid developers in selecting this lowest level Marmot calculates the minimal required thread level at runtime.
- **Correct usage of shared memory:** Applications that use the highest thread level may call MPI in parallel. Many MPI calls assume that memory passed to MPI is owned by it for a certain amount of time. Such memory must not be touched as long as the memory is owned by MPI. When multiple threads execute MPI calls in parallel this restriction is easily violated. Such an example is shown below.

```
#pragma omp parallel
{
  MPI_Recv(my_buf,...)
}
```

  In this example the shared variable "my_buf" is passed to MPI multiple times. These errors are currently not detected by Marmot. In order to detect all instances of these problems it is necessary to detect every access to memory owned by MPI. This might be achieved by an own implementation or by using existing tools like Valgrind [24]. In order to detect a subset of these errors it is possible to check whether memory passed to MPI is already owned by MPI due to a preceding call.
- **Conformance to special restrictions:** The MPI standard documents mention several special restrictions for multi-threaded usage. Restrictions refer to various parts of the interface: examples are initializing and finalizing MPI, usage of communicators in collective calls, usage of requests and message probing. For the above mentioned restriction stating that each communicator must not be used in multiple collective calls simultaneously, we want to present a simple example:

```
#pragma omp parallel
{
  MPI_Barrier(MPI_COMM_WORLD)
}
```

  This code snippet violates the restriction for the communicator `MPI_COMM_WORLD`. It is necessary to create special checks for each restriction which is currently done for all identified MPI-1 restrictions. These checks detect violations if they actually occur in a run made with Marmot.

4 Collaboration with other tools

4.1 Marmot and CUBE

By default, Marmot prints its errors, warnings and remarks into a human-readable text file. Another option is HTML logging which results in an HTML file that can be viewed with a standard browser. However, both of these output formats result in a chronological list of events. To provide also an hierarchical view of the messages, Marmot can make use of the CUBE library included in the KOJAK (Scalasca resp.) toolset [14, 15]. In this case, the output is written to an XML file which can then be viewed with KOJAK's visualizer CUBE. An example of Marmot's log file in XML format visualized with CUBE is depicted in Fig. 3. One can see that the CUBE

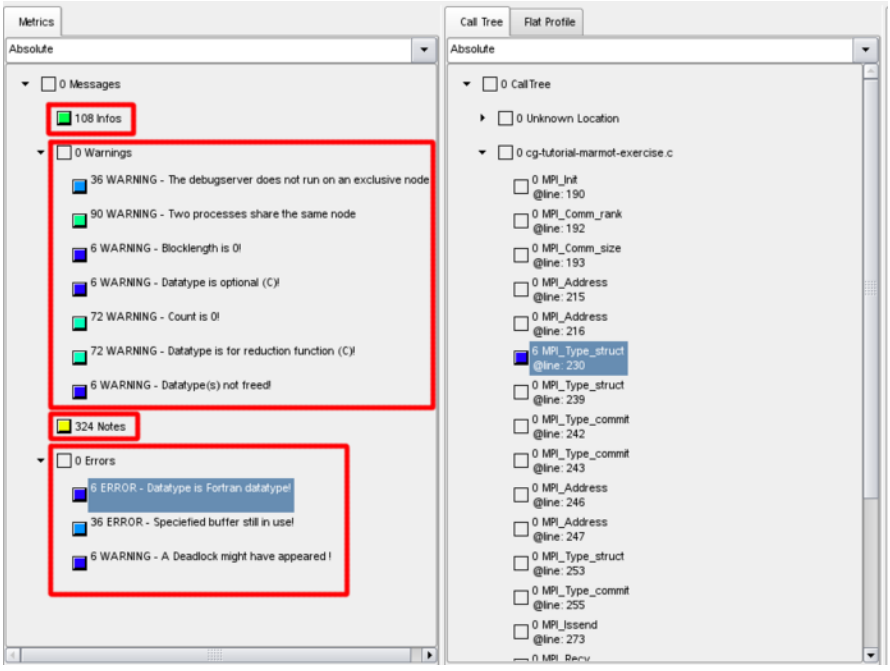


Fig. 3 Visualisation with CUBE (display detail)

visualizer presents Marmot's messages in a hierarchical tree view. Infos, warnings, notes and errors are not displayed in chronological order but are grouped, so that a user can easily identify e.g. only errors. This visualizer provides an intuitive way of browsing through the Marmot messages. An improved version (CUBE3) is about to be released and will also be supported by Marmot.

4.2 Marmot and DDT

Until now, Marmot has generally been used as a standalone tool. This is about to change with the integration into the Distributed Debugging Tool (DDT) from Allinea [16]. DDT is a source level debugger for C, C++ and Fortran. It supports practically all implementations of MPI, OpenMP and combinations thereof (MPI/OpenMP hybrid). DDT provides a convenient graphical user interface that meets the demands of parallel debugging. To combine the strength of the debugger with Marmot's ability of runtime MPI correctness checking, a plugin for DDT is under development [13]. The user will then be able to activate or deactivate Marmot on a per run basis. Furthermore, DDT's graphical user interface will be used to provide for a user friendly configuration of the Marmot plugin, e.g. settings concerning Marmot's log level (Info, Warning, Error) can then be modified using an intuitive GUI, instead of modifying the respective environment variables via command line. DDT will also, on the one hand, take care of starting a run with Marmot with an additional process for the debug server and, on the other hand, hide this process from the user to allow for a consistent way of working with the debugger. DDT's graphical user interface will also be used to display Marmot's messages. This is especially useful as messages with different severity may be displayed in different panes and a user's click on such a message will jump to the appropriate line in the source code. One can also make DDT pause the program in case Marmot detects an error. This way, the user can step through the program using both the debugger and the correctness checker at the same time.

4.3 Marmot and Intel® Thread Checker

Some of the additional checks presented in Sect. 3.2 that are used for hybrid applications will only detect errors if they actually appear in a run made with Marmot. For some applications this might be a problem as the probability of an error to occur might be very low. As an simplified example we present the following code snippet:

```
#pragma omp parallel private(thread)
{
    #pragma omp sections
    {
        #pragma omp section
        { MPI_Barrier(MPI_COMM_WORLD); }
        #pragma omp section
        { sleep(5);
          MPI_Barrier(MPI_COMM_WORLD); }
    }
}
```

This example violates the collective communicator restriction of the MPI standard for some very improbable runs. Thus detecting this error with Marmot is almost impossible, as usual runs will not contain the error.

In order to improve detection of these errors it is possible to use Marmot in combination with Intel® Thread Checker [27]. Additional code executed in Marmot makes the Thread Checker aware of violations to MPI restrictions. This is achieved by creating artificial data races that only occur if such a restriction is violated. The output of the Thread Checker contains data race errors if a restriction is violated.

4.4 Marmot and Visual Studio on Windows

The Windows Version of Marmot was tested and works well with MPIch2 and MSMPI (contained in the Compute Cluster Pack). MSMPI is based on the reference MPIch2 implementation and differs mainly in job launch and management due to security considerations made in Windows Compute Cluster Server. Although there are precompiled versions of Marmot for the two MPI implementations a user might still be forced to recompile it. Compiling for the specific version of the MPI implementation and runtime used is highly recommended. In order to configure/build Marmot the cmake tool [21] is needed. On Windows it is imperative to avoid mixing debug and release builds of libraries. Therefore a 'D' suffix is added to the debug builds of the Marmot libraries. Users could manually adjust their existing Visual Studio projects and link to the Marmot libraries or simply use an example `CMakeLists.txt` file from the Marmot repository and adjust that to their needs.

Since it can be quite tedious to lookup errors and warnings in log-files an AddIn was developed to better integrate Marmot into Visual Studio. The AddIn launches the application selected as 'Startup Project' in Visual Studio and communicates with the Debug-Server built into Marmot (see Fig. 1). The Marmot output is displayed similarly to compile warnings and errors in the Output pane of Visual Studio (see Fig. 4).

5 Experiences with real Applications

Marmot has been used with synthetic test programs but also with a number of real-world applications and benchmarks.

5.1 Bloodflow Simulation

There are many examples of errors that are tolerated by MPI implementations or that only occur on specific platforms, or occur under specific circumstances. When

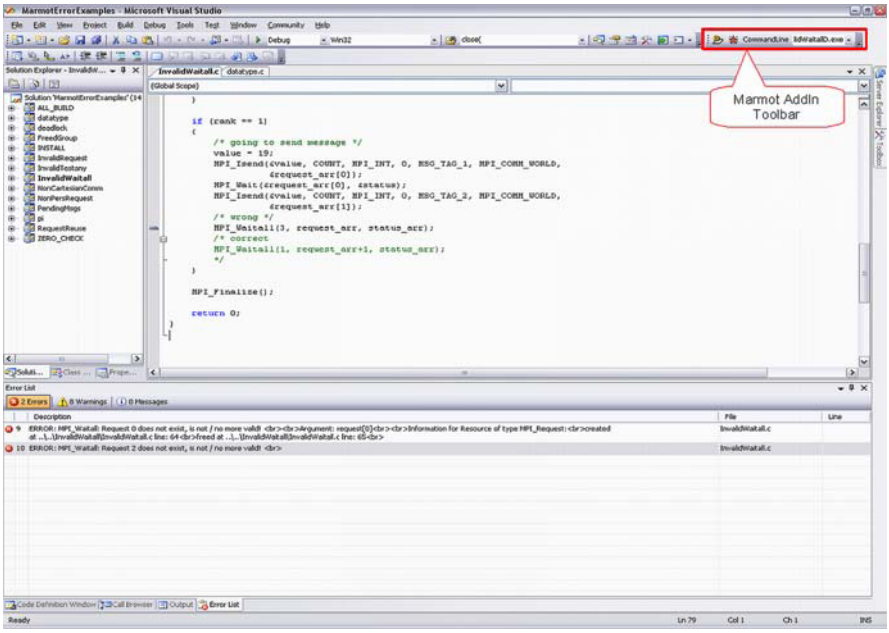


Fig. 4 Marmot Visual Studio AddIn

analysing the development version of a medical application that uses a 3D Lattice-Boltzmann method for blood flow calculation, we find different problems in the code. In many places the developers equate `MPI_Comm` with `int`. This is a dangerous thing to do because, in MPIch, the opaque object `MPI_Comm` is actually defined as an `int` and therefore the code works without a problem. However, in LAM/MPI, `MPI_Comm` is defined as a pointer to a `struct` and therefore it breaks on any platform where a pointer does not fit into an integer.

When we test this application with different input files representing the geometry of the artery we find other problems. In the simplest case, a mere tube with an approximately constant radius, the code runs without any problems. When calculating the blood flow for an artery stenosis, i.e. using a tube with varying radius, the application stalls and Marmot finds a deadlock caused by process 0, which performs an `MPI_Sendrecv` whereas all other processes perform an `MPI_Bcast`. A very simplified skeleton of the source code shows why:

```
main {
  ...
  //compute number of iterations depending on the radius
  if (radius < x) num_iter = y; else num_iter = z;

  for (i=0; i < num_iter; i++)
  {
```

```
    // compute blood flow and exchange results
    // with neighbours using MPI_Sendrecv
    computeBloodflow(...);
  }

  // communicate results using MPI_Bcast
  writeResults(...);
}
```

Every process calculates its own number of iterations depending on the radius, but unfortunately, they do not communicate to agree on a maximum number of iterations. As a result, process 0 tries to perform more iterations having a piece of the artery with a bigger radius than the others, and therefore tries to exchange results with its neighbour using the `MPI_Sendrecv` whereas the others already have finished their iterations and try to communicate with the `MPI_Bcast`. This shows how important it is to choose relevant input data sets for an effective runtime checking. It also shows how difficult it is for developers to keep track of all the MPI communication when it is hidden in subroutines.

Another input file representing a forked artery reveals yet another programming error. In this case, every process results in different values for the send/receive counts in the collective call `MPI_Gather`. On some platforms the application runs without a problem, but on some platforms the different values cause a segmentation violation. Strangely enough, on one platform the application runs without a problem, but when attaching a performance analysis tool to it, it crashes. It appears to be the fault of the tool, but in reality there is a bug in the application. Antithetically, it is possible that bugs never occur in the presence of tools (so-called).

5.2 SPEC MPI2007 Benchmarks

SPEC MPI2007 [34] is a suite of applications used to evaluate performance of parallel computing systems. A discussion on one of the SPEC MPI2007 mailinglists led to the assumption that there was a bug in one of the benchmarks. The discussion started as invalid results with a certain MPI implementation appeared. Members of the mailing list identified the existence of multiple uncompleted `MPI_Irecv` calls that used the same receive buffer. According to the MPI standard this is erroneous.

We used Marmot to confirm the existence of the problem and to validate the correctness of possible solutions. A first run with Marmot resulted in a segmentation fault. At first we assumed that there was an error in Marmot but eventually we found out that there was a second bug in the application. The second error resulted from a `MPI_Irecv` call which is drafted below.

```
call MPI_IRECV( buf, size, type,     &
                source, tag, comm,   &
                request, status, error)
```

This call contains the superfluous argument *status*. Detecting this usually is a compiler task but, due to the usage of Fortran77, this was not possible. When using Marmot additional arguments are appended to the MPI calls and, due to the superfluous parameter, this led to the segmentation fault.

After fixing this error we were able to execute the application with Marmot. The logfiles confirmed the existence of uncompleted `MPI_Irecv` calls that use the same receive buffers. Such a situation arises when executing code like:

```
MPI_Irecv(buf, ... );
MPI_Irecv(buf, ... );
```

Three different solutions were proposed to solve this error. Marmot confirmed that two of the solutions solved the problem. The third solution corrected the problem by ensuring that only one of the `MPI_IRecv` calls is satisfied at a time. With this solution Marmot still detected that memory owned by MPI is used in another MPI call. When interpreting the MPI standard strictly this is still a usage error of the MPI, however, as common MPI implementations will only touch the buffer when the receive is satisfied this will not cause any actual errors.

5.3 Spin Glass application

Marmot is integrated in the Interactive European Grid infrastructure [31].

The Spin Glass application [32, 33] is a physics application that is used to examine the effects of temperature changes on the physical characteristics of spin glass - mainly on magnetization. The Spin Glass application uses MPI communication with both collectives and point-to-point communication. There was a phase in the development of the application when the program was hanging. Marmot delivered a hint for wrong data types being used in the MPI communication. The developer had a look into the log file to examine the order in which the calls were made. Although the log did not identify the problem, the list of MPI calls helped him to trace the error as a race condition. The code was then corrected.

The tests done by project members resulted in some very valuable feedback for the developers. Improvements need to be done in the deadlock detection mechanism, as sometimes the high latency and low bandwidth of a cluster of workstations result in increased pending times for communication.

6 How to install and use Marmot

For the readers who are interested in using Marmot themselves we give a very short description of how to install and use it. A download version and installation instructions can be found on Marmot's website [9]. Once the installation is done the usage of Marmot is quite easy. Two steps have to be performed: First, compilation of the

application and linking it against the Marmot libraries. Second, running the program with an additional process (needed for Marmot's debug server). For the first step, an installation of Marmot provides compiler wrappers (*marmotcc, marmotcxx* and *marmotf77*) which do the necessary linking. Suppose the user wants to run *SomeApp* with Marmot attached and originally uses 3 processes. Then one would typically issue the commands

```
marmotcc -o SomeApp SomeApp.c
mpirun -np 4 ./SomeApp
```

After the run a file named *Marmot_SomeApp_[TIMESTAMP].txt* can be found in the working directory. This file contains Marmot's output. If the user wants to view the results in a browser the logging format can be switched to HTML by modifying the environment variable *Marmot_LOGFILE_TYPE*:

```
export Marmot_LOGFILE_TYPE=1
```

An excerpt of such an HTML file is depicted in Fig. 2. An overview on Marmot's environment variables can be found in the userguide or in the log file's header.

7 Conclusion and Future Work

In this paper we have presented the Marmot MPI correctness checker, which analyses the behaviour of an MPI application during runtime and checks for errors frequently made in the use of the MPI API. The functionality of this tool has been tested successfully with real world applications.

Future work includes technical improvements, e.g. a better deadlock detection mechanism or full support for the MPI-2 standard. Another aspect is to improve the performance and scalability of the tool, especially for communication-intensive applications. We also aim at constantly improving user-friendliness, e.g. by adapting CUBE visualisation better to the needs of Marmot's correctness checking messages.

Acknowledgements The research presented in this paper has partially been supported by the Federal Ministry of Research and Education (BMBF) through the ITEA2-06015 project "ParMA" [30] (June 2007 – May 2010), by the Virtual Institute - High Productivity Supercomputing (VI-HPS), which is funded by the Helmholtz Association under Grant No. VH-VI-228, by the European Union through the IST-031857 project "int.eu.grid" (May 2006 – April 2008) and by Microsoft.

References

1. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. http://www.mpi-forum.org/.
2. Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. http://www.mpi-forum.org/.

3. Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference (SC 2000)*, Dallas, Texas, 2000.
4. William D. Gropp. Runtime Checking Of Datatype Signatures In MPI. In *Recent Advances In Parallel Virtual Machine And Message Passing. 7th European PVM/MPI Users' Group Meeting. LNCS 1908*, pages 160-167. Springer 2000.
5. Chris Falzone, Anthony Chan, Ewing Lusk and William Gropp. Collective Error Detection for MPI Collective Operations. In *Recent Advances In Parallel Virtual Machine And Message Passing. 12th European PVM/MPI Users' Group Meeting. LNCS 3666*, pages 138-147. Springer 2005.
6. J.L. Träff and J. Worringen. Verifying Collective MPI Calls. In *Recent Advances In Parallel Virtual Machine And Message Passing. 11th European PVM/MPI Users' Group Meeting. LNCS 3241*, pages 18 - 27, Springer, 2004.
7. Dieter Kranzlmüller. *Event Graph Analysis For Debugging Massively Parallel Programs.* Phd thesis, Joh. Kepler University Linz, Austria, 2000.
8. Glenn Luecke, Yan Zou, James Coyle, Jim Hoekstra and Marina Kraeva. Deadlock Detection In MPI Programs. In *Concurrency and Computation: Practice and Experience.* 2002, vol. 14, pages 911 - 932.
9. Marmot. http://www.hlrs.de/organization/amt/projects/marmot
10. Bettina Krammer, Matthias S. Müller and Michael M. Resch. MPI I/O Analysis and Error Detection with Marmot. In *Recent Advances In Parallel Virtual Machine And Message Passing. 11th European PVM/MPI Users' Group Meeting. LNCS 3241*, pages 242 - 250, Springer, 2004.
11. Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. Marmot: An MPI analysis and checking tool. In *Proceedings of PARCO 2003*, pages 493-500, Elsevier, 2004.
12. Bettina Krammer, Matthias S. Müller and Michael M. Resch. MPI Application Development Using the Analysis Tool Marmot, In *Proceedings of ICCS 2004*, *LNCS 3038*, pages 464 - 471, Springer 2004.
13. Bettina Krammer, Valentin Himmler, David Lecomber. Coupling DDT and Marmot for Debugging of MPI Applications. In *Proc. of ParCo 2007*, Jülich/Aachen, Germany, September 4-7, 2007. NIC Series, Vol. 38, pp. 653-660
14. KOJAK. Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks http://www.fz-juelich.de/jsc/kojak/
15. Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable Parallel Trace-Based Performance Analysis. In *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference*, *LNCS 4192*, pages 303 - 312, Springer 2006.
16. DDT. The Distributed Debugging Tool. http://www.allinea.com/?page=48
17. Totalview. http://www.totalviewtech.com/productsTV.htm
18. mpigdb. http://www-unix.mcs.anl.gov/mpi/MPICH/docs/userguide/-node26.htm#Node29
19. The GNU Project Debugger. http://www.gnu.org/manual/gdb
20. The Data Display Debugger. http://www.gnu.org/software/ddd/
21. The Cross-Platform Makefile Generator http://www.cmake.org
22. Brett Carson and Ian A. Mason. ClusterGrind: Valgrinding LAM/MPI Applications. In *Recent Advances In Parallel Virtual Machine And Message Passing. 12th European PVM/MPI Users' Group Meeting. LNCS 3666*, pages 325-332. Springer 2005.
23. Rainer Keller, Shiqing Fan and Michael Resch. Memory debugging of MPI-parallel Applications in Open MPI. In *Proceedings of ParCo'07*, G.R. Joubert et al. (eds), Juelich, Germany, September, 2007.
24. Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, USENIX Association (2005), 2–2.

25. Jayant DeSouza, Bob Kuhn and Bronis R. de Supinski. Automated, scalable debugging of MPI programs with Intel Message Checker. *SE-HPCS '05*, St. Louis, Missouri, USA. http://csdl.ics.hawaii.edu/se-hpcs/papers/11.pdf
26. Patrick Ohly and Werner Krotz-Vogel. Automated MPI Correctness Checking: What if There Were a Magic Option? *8th LCI '07*, South Lake Tahoe, California, USA. May 2007. http://softwarecommunity.intel.com/isn/Downloads/multicore/Krotz-Vogel_lci-hpcc-correctness.pdf
27. Sack, P., Bliss, B.E., Ma, Z., Petersen, P., Torrellas, J.: Accurate and efficient filtering for the intel thread checker race detector. In: ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability, New York, NY, USA, ACM (2006) 34–41
28. A. Tirado-Ramos, H. Ragas, D. Shamonin, H. Rosmanith, and D. Kranzlmueller. Integration of blood flow visualization on the grid: the flowfish/gvk approach. In *2nd European Across Grids Conference*, Nicosia, Cyprus, January 28-30 2004.
29. ParMA: Parallel Programming for Multi-core Architectures - ITEA2 Project (06015). http://www.parma-itea2.org/
30. Bettina Krammer and Rainer Keller. The ParMA Project. inSiDE, Vol 5, No. 1, Spring 2007.
31. Interactive European Grid. http://www.interactive-grid.eu/
32. S. Jimenez, V. Martin-Mayor, S. Perez-Gaviro. Rejuvenation and Memory in model Spin Glasses in 3 and 4 dimensions. Phys. Rev. B 72, 054417 (2005).
33. I. Campos, M. Cotallo-Aban, V. Martin-Mayor, S. Perez-Gaviro, A. Tarancon. Phys. Rev. Lett. 97, 217204 (2006).
34. M. S. Müller, M. van Waveren, R. Liebermann, B. Whitney, H. Saito, K. Kalyan, J. Baron, B. Brantley, Ch. Parrott, T. Elken, H. Feng and C. Ponder SPEC MPI2007 - An Application Benchmark for Clusters and HPC systems *In Proceedings of ISC2007*, Dresden, 2007.

Memory Debugging in Parallel and Distributed Applications

Chris Gottbrath

Abstract Memory errors, such as memory leaks and bounds violations are often the source of the kind of bugs that are especially challenging for scientists, computer scientists and engineers to resolve. This paper describes a new software development tool called MemoryScape that provides developers with a highly graphical and interactive memory debugging tool. MemoryScape can be used to troubleshoot problems on applications ranging from serial applications for the desktop or server up to massively multprocess applications running on supercomputers. This paper provides an introduction to some of the challenges of memory debugging in parallel architectures, reviews the memory errors detected and provides an overview of the Heap Interposition Agent (HIA) and parallel debugging technology that makes this possible.

1 Introduction

Memory bugs, essentially a mistake in the management of heap memory, can occur in any program that is being written, enhanced, or maintained. A memory bug can be caused by a number of factors, including failure to check for error conditions; relying on non-standard behavior; memory leaks including failure to free memory; dangling references such as failure to clear pointers; array bounds violations; and memory corruption such as writing to memory not owned / over running array bounds. These can sometimes cause programs to crash or generate incorrect "random" results, or more frustratingly, they may lurk in the code base for long periods of time - only to manifest at the worst possible time.

Memory problems are difficult to track down with conventional tools on even a simple desktop architecture, and are much more vexing when encountered on a

Chris Gottbrath
TotalView Technologies, 24 Prime Parkway, Natick, MA 01760, e-mail: Chris.Gottbrath@totalviewtech.com

distributed parallel architecture. This paper will review the challenges of memory debugging, with special attention paid to the challenges of parallel development and parallel debugging, and introduce a tool that helps developers identify and resolve memory bugs in parallel and distributed applications, highlight major features, and discuss architectural choices so that users can understand benefits and drawbacks of some of those choices.

2 The Challenges of Memory Debugging in Parallel Development

The fact that memory bugs can be introduced at any time makes memory debugging a challenging task- especially in codes that are written collaboratively or that are being maintained over a long period of time, where assumptions about memory management can either change or not be communicated clearly. They can also lurk in a code base for long periods of time since they are often not immediately fatal and can suddenly become an issue when a program is ported to a new architecture, scaled up to a larger problem size, or when code is adapted and reused from one program to another.

Memory bugs often manifest themselves in several ways, either as a crash that always happens, a crash that sometimes happens (instability), or just as incorrect results. Furthermore, they are difficult to track down with commonly used development tools and techniques, such as printf and traditional source code debuggers, which are not specifically designed to solve memory problems.

Adding parallelism to the mix makes things even harder because parallel programs are often squeezed between two effects, meaning that these programs have to be very careful with memory. Parallel programs are also written in situations where the problem set is "large," so the program naturally ends up loading a very significant amount of data and using a lot of memory. However, special purpose HPC systems often have less memory per node than one might ideally desire, as memory is expensive.

3 Classifying Memory Errors

Programs typically make use of several different categories of memory that are managed in different ways. These include stack memory, heap memory, shared memory, thread private memory and static or global memory. However, programmers are required to pay special attention to memory that is allocated out of the heap memory. This is because the management of heap memory is done explicitly in the program rather than implicitly at compile or run time.

There are a number of ways that a program can fail to make proper use of dynamically allocated heap memory. It is useful to develop a simple categorization of these mistakes for discussion; in this paper, they will be described in terms of the C mal-

loc() API. However, it is important to note that analogous errors can also be made with memory that is allocated using the C++ new statement and the FORTRAN 90 allocate statement.

3.1 Malloc Errors

Malloc errors occur when a program passes an invalid value to one of the operations in the C Heap Manager API. This could potentially happen if the value of a pointer (the address of a block) was copied into another pointer, and then at a later time, both pointers were passed to free(). In this case, the second free() is incorrect because the specified pointer does not correspond to an allocated block. The behavior of the program after such an operation is undefined.

3.2 Dangling Pointers

A pointer can be said to be dangling when it references memory that has already been deallocated. Any memory access, either a read or a write, through a dangling pointer can lead to undefined behavior. Programs with dangling pointer bugs may sometimes appear to function without any obvious errors, even for significant amounts of time, if the memory that the dangling pointer points to happens not to be recycled into a new allocation during the time that it is accessed.

3.3 Memory Bounds Violations

Individual memory allocations that are returned by malloc()represent discrete blocks of memory with defined sizes. Any access to memory immediately before the lowest address in the block or immediately after the highest address in the block results in undefined behavior.

3.4 Read-Before-Write Errors

Reading memory before it has been initialized is a common error. Some languages assign default values to uninitialized global memory, and many compilers can identify when local variables are read before being initialized. What is more difficult and generally can only be done at random is detecting when memory accessed through a pointer is read before being initialized. Dynamic memory is particularly affected,

since this is always accessed through a pointer, and in most cases, the content of memory obtained from the memory manager is undefined.

4 Detecting Memory Leaks

Leaks occur when a program finishes using a block of memory, discards all references to the block, but fails to call free()to release it back to the heap manager for reuse. The result is that the program is neither able to make use of the memory nor reallocate it for a new purpose.

The impact of leaks depends on the nature of the application. In some cases the effects are very minor; in others, where the rate of leakage is high enough or the runtime of the program is long enough, leaks can significantly change the memory behavior and the performance characteristics of the program. For long running applications or those where memory is limited, even a small leakage rate can have a very serious cumulative and adverse effect. This somewhat paradoxically makes leaks all that much more annoying – since they often linger in otherwise well-understood codes. It can be quite challenging to manage dynamic memory in complex applications to ensure that allocations are released exactly once so that malloc and leak errors do not occur.

Leak detection can be done at any point in program execution. As discussed, leaks occur when the program ceases using a block of memory without calling free. It is hard to define “ceasing to use” but an advanced memory debugger is able to execute leak detection by looking to see if the program retains a reference to specific memory locations.

5 The MemoryScape Debugger

The MemoryScape memory debugger is an easy-to-use tool for developers to get started using. It has a lightweight architecture that requires no recompilation and has modest impact on the runtime performance of the program. The interface is designed around the concept of an inductive user interface, which guides the user through the task of memory debugging and provides easy-to-understand graphical displays, powerful analysis tools, and features to support collaboration (making it easy to report a lurking memory bug to the library vendor, scientific collaborator, or colleague who wrote the code in question).

MemoryScape is designed to be used with parallel and multiprocess target applications, providing both detailed information about individual processes, as well as high level memory usage statistics across all of the processes that make up a large parallel application. MemoryScape’s specialized features, including support for launching and automatically attaching to all of the processes of a parallel job, the ability to memory debug many processes from within one GUI and the ability to

do script-based debugging to use batch queue environments, make it well-suited for debugging these parallel and distributed applications.

6 MemoryScape Architecture

MemoryScape accomplishes memory debugging on parallel and distributed applications through the modified use of a technique called interposition. MemoryScape provides a library, called the Heap Interposition Agent (HIA), that is inserted between the user's application code and the malloc()subsystem. This library defines functions for each of the memory allocation API functions. It is these functions that are initially called by the program whenever it allocates, reallocates, or frees a block of memory.

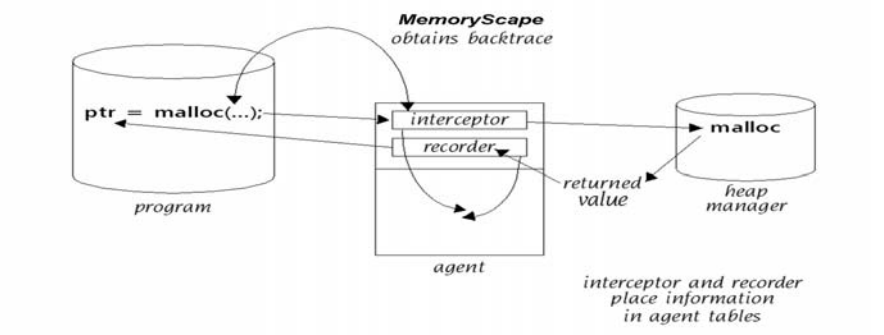


Fig. 1 MemoryScape Heap Interposition Agent (HIA) Architecture. The HIA sits between the application and the memory allocation layer in glibc

The interposition technique used by MemoryScape was chosen in part because it provides for lightweight memory debugging. Low overheads are an important factor if the performance of a program is not to suffer because of the presence of the HIA. In most cases, the runtime performance of a program being debugged with the HIA engaged will be similar to that where the HIA is absent. This is absolutely critical for high-performance computing applications, where a heavyweight approach that significantly slowed the target program might very well make the runtime of programs exceed the patience of developers, administrators and job schedulers.

Interposition differs from simply replacing the malloc() library with a debug malloc in that the interposition library does not actually fulfill any of the operations itself – it arranges for the program's malloc() API function calls to be forwarded to the underlying heap manager that would have been called in the absence of the HIA. The effect of interposing with the HIA is that the program behaves in the same way that it would without the HIA- except that the HIA is able to intercept all of the

memory calls and perform bookkeeping and "sanity checks" before and after the underlying function is called.

The bookkeeping that the HIA library does builds up and maintains a record of all of the active allocations on the heap as the program runs. For each allocation in the heap, it records not just the position and size of the block, but also a full function call stack representing what the program was doing when the block was allocated. The "sanity checks" that the HIA performs are the kinds of things that allow the HIA to detect malloc() errors such as freeing the same block of memory twice or trying to reallocate a pointer that points to a stack address.

Depending on how it has been configured, the HIA can also detect whether some bounds errors have occurred. The information that the HIA collects is used by the MemoryScape memory debugger to provide the user with an accurate picture of the state of the heap.

6.1 MemoryScape Parallel Architecture

MemoryScape uses a behind-the-scenes, distributed parallel architecture to manage runtime interaction with the user's parallel program. MemoryScape starts lightweight debugging agent processes (called tvdsvr processes for historical reasons), which run on the nodes of the cluster where the user's code is executing. These tvdsvr processes are each responsible for the low level interactions with the individual local processes and the HIA module that is loaded into the process that is being debugged. The tvdsvr processes communicate directly with the MemoryScape front-end process, using their own optimized protocol, which in most cluster configurations is layered on top of TCP/IP.

7 MemoryScape Features

7.1 Using MemoryScape to Compare Memory Statistics

Many parallel and distributed applications have known or expected behaviors in terms of memory usage. They may be structured so that all of the nodes should allocate the same amount of memory, or they may be structured so that memory usage should depend in some way on the MPI_COMM_WORLD rank of the process. If such a pattern is expected or if the user wishes to simply examine the set of processes to look for patterns, MemoryScape features a memory statistics window that provides overall memory usage statistics in a number of graphical forms (line, bar and pie charts) for one, all, or an arbitrary subset of the processes that make up the debugging session. The user may drive the program to a specific breakpoint or barrier, or simply halt all the processes at an arbitrary point in execution.

The set of processes that the user wishes to see statistical information about may be selected, with the type of view that the user wants, by clicking "generate view." The generated view represents the state of the program at that point in time. The user may use the debugger process controls to drive the program to a new point in execution and then update the view to look for changes. If any processes look out of line, the user will likely want to look more closely at the detailed status of the heap memory.

7.2 Using MemoryScape to Look at Heap Status

MemoryScape provides a wide range of heap status reports, the most popular of which is the heap graphical display. At any point where a process has been stopped, a user can obtain a graphical view of the heap. This is obtained by selecting the heap status tab, selecting one or more processes, choosing the graphical view and clicking "generate view."

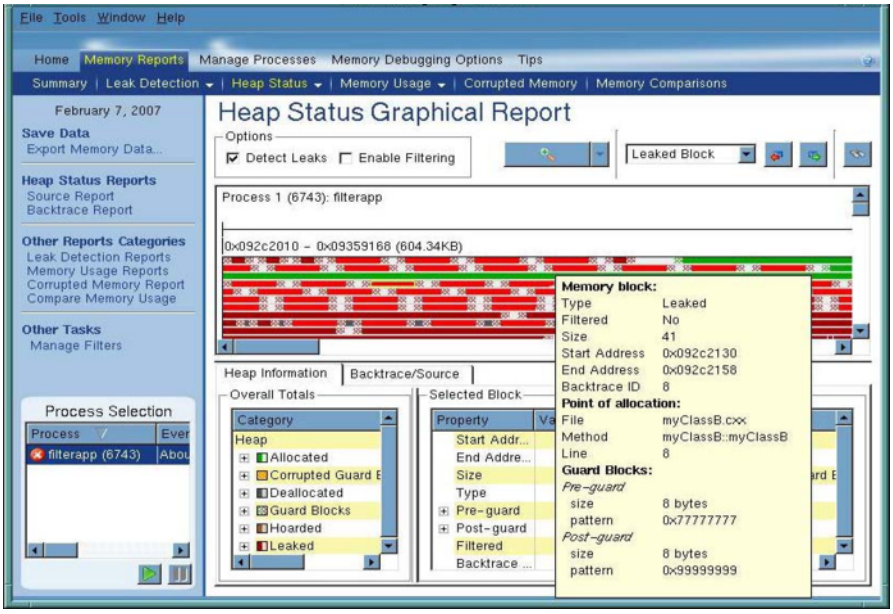


Fig. 2 MemoryScape Graphical Interface provides an interactive view of the heap. Colors indicate the status of memory allocations

The resulting display paints a picture of the heap memory in the selected process. Each current heap memory allocation is represented by a green line extending across the range of addresses that are part of the allocation. This gives the user a great way to see the composition of the program's heap memory at a glance. The view

is interactive; selecting a block highlights related allocations and presents the user with detailed information about both the selected block and the full set of related blocks. The display can be filtered to dim allocations based on properties such as size or the shared object they were allocated in. The display also supports setting a baseline to let the user see which allocations and deallocations occur before and after that baseline.

7.3 Using MemoryScape to Detect Leaks

MemoryScape performs heap memory leak detection by driving the program to a known state (a breakpoint, for example) or by simply halting the processes of a running parallel application using the "halt" command in the GUI or the CLI. By selecting the leak detection tab in the memory debugging window, one or more of the processes in the parallel job can be selected to generate the leak report.

The resulting report will list all of the heap allocations in the program for which there are no longer any valid references anywhere in the program's registers, or accessible memory. A block of memory that the program is not storing a reference to anywhere is highly unlikely to subsequently be subject to a free() call and is extremely likely to be a leak. Leaks can also be observed in the heap graphical display discussed above by toggling the checkbox labeled "Detect Leaks" in which leaked blocks will be displayed in red on the graphical display.

7.4 Using MemoryScape to Detect Heap Bounds Violations

One of the classes of memory errors mentioned above that is often difficult to diagnose is when an error in the program logic causes the program to write beyond the bounds of a block of memory allocated on the heap. The malloc API makes no guarantee about the relative spacing or alignment of memory blocks returned in separate memory allocations – or about what the memory before or after any given block may be used for. The result of reads and writes before the beginning of a block of memory, or after the end of the block of memory, is undefined.

In practice, blocks are often contiguous with other blocks of program data. Therefore, if the program writes past the end of an array, it is usually overwriting the contents of some other unrelated allocation. If the program is re-run and the same error occurs, the ordering of allocations may differ and the overwriting may occur in a different array. This leads to extremely frustrating "racy" bugs that manifest differently, sometimes causing the program to crash, sometimes resulting in bad data, and sometimes altering memory in a way that turns out to be completely harmless.

MemoryScape provides a mechanism that involves setting aside a bit of memory before and after heap memory blocks as they are allocated. Since this bit of memory, called a guard block, is not part of the allocation, the program should never read or

write to that location. The HIA can arrange for the guard blocks to be initialized with a pattern and check the guard blocks (any time the user asks for a check and again any time an individual block is deallocated) for a change in this pattern. Any changes mean that the program wrote past the bounds of the array.

7.5 Collaboration Features in MemoryScape

MemoryScape provides users with two forms of memory reporting that help distributed development teams collaborate effectively to troubleshoot problems and improve product quality. Heap memory views can be exported as an HTML file that can be read by a web browser. These HTML files include Javascript applets that provide the user with the ability to interact with the report in their browser in a similar way to the way that the report can be interacted with on screen – allowing the reader to drill down into sections of the report that are interesting and survey the rest of the report at a summary level.

MemoryScape also supports the creation of a memory debugging data file. This is a binary representation of all of the data that MemoryScape has about a process. These files can be loaded back in at a later date and interacted with just like live processes. This gives developers the ability to store representations of processes for later comparison and examination.

Memory debugging data files can also be loaded by the memory module of the TotalView™ Source Code Debugger. This allows sophisticated users to apply even more powerful and precise debugging techniques that take advantage of the idea of having memory debugging and access to all the variables and state data for live processes.

8 MemoryScape Usage Tips

As discussed, MemoryScape detects many instances where a program has erroneously written outside the bounds of heap arrays. Developers and scientists can compliment MemoryScape's heap bounds checking with compiler-generated bounds checking code for arrays that are allocated automatically on the heap or in global program memory. A number of compilers, including the Intel™ Fortran Compiler and the open source gfortran compiler, can generate bounds checking code automatically with a compile line option, for example (-check-bounds for ifort and -fbounds-check for gfortran).

For developers with more advanced memory debugging needs, the TotalView Debugger provides a way of debugging memory problems that allows the user to examine memory information within the source code context of the program. When doing memory debugging using the TotalView source code debugger, the user can examine data structures that might contain pointers into the heap memory within the

program. When memory debugging is enabled, these pointers are annotated with information about the status of the memory block being pointed to.

One advanced technique available to users who are using the TotalView source code debugger in conjunction with the memory debugger involves the use of watchpoints. Watchpoints are a debugger feature that allows the process to be stopped at the moment when a given block of memory is written to. When a pointer is writing "wildly" across memory (into space that is completely unrelated to where the pointer is supposed to be writing), it can be very hard to pin down.

Guard blocks can be used together with watchpoints to track down this exceptionally subtle type of error. The troubleshooting takes two passes. On the first pass through the program, guard blocks are used to identify a specific block of memory that is erroneously written to. Then, on the second pass through the program, a watchpoint is set at that precise address. The watchpoint should trigger twice: once when the block of memory is painted by the memory debugger and the second time when the block of memory is overwritten by the wild pointer.

While most users will want to use MemoryScape interactively as outlined above, ongoing development introduces the possibility that new memory bugs may be introduced at any point. Development teams are encouraged to add heap memory tests to their ongoing testing strategy. MemoryScape includes a non-interactive command line version that is specifically designed to be incorporated into an automatic testing framework. Development teams that use MemoryScape in this way can detect, analyze and remove new memory errors as soon as they are introduced in development – before they have any impact in production.

9 MemoryScape User Case Study: SIMULIA Uses MemoryScape to Find and Fix Bugs Quickly

DASSAULT SYSTEMES is a worldwide PLM leader and software innovator. The company's SIMULIA software solutions empower users to create, share and experience in 3D. SIMULIA's scalable portfolio of realistic simulation solutions improves product performance, reduces physical prototypes and drives innovation, including the CATIA Analysis applications, the Abaqus product suite for Unified Finite Element Analysis, multiphysics solutions for insight into challenging engineering problems, and lifecycle management solutions for managing simulation data, processes and intellectual property.

Tremendous attention and resources are dedicated to ensuring the high quality of SIMULIA products. Nevertheless, with each release a few "mysterious" problems may creep in. They occur very rarely (once in about every 100 or even 1000 runs), but when they do occur, they defy all human efforts to capture them. These few problems could also potentially draw tremendous resources.

Historically, the task of finding and fixing such memory bugs has been a time-consuming and complex task; finding and then understanding these bugs would take a considerable amount of effort and was very expensive because it involved a great

deal of human time and attention. The problems would occur in fully optimized builds when run on loaded machines, where there is strong contention for memory between processes, and would also usually occur in very long running jobs that often take days. Additionally, in many cases there would be no core-dump. Even in the cases where there was a core-dump, it would point to the result of the corruption, not the source of it. The source may often be quite a distance away from the point of the crash.

In most cases, it was found that these mysterious and elusive problems were being caused by subtle memory problems that would flee at the moment's notice. After many frustrating attempts to diagnose these issues using logic and reasoning, it became clear that SIMULIA developers needed a better way to gain insight into what was happening with memory. Thus the search for a proper tool began. SIMULIA was looking for something that would allow them to intercept such problems from the start, to capture their cause, and to fix them at their origin. They ultimately chose to employ the MemoryScape memory debugger from TotalView Technologies.

Since the company began using the MemoryScape, developers have been able to find memory problems easily and fix them very quickly - with little effort. "MemoryScape accelerates our ability to identify where and why problems occur in our software," said Nick Monyatovsky, software engineer at SIMULIA. "When a problem occurs, MemoryScape's GUI provides a very clear view of the source of the problem, and its scripting interface, and the tool allows us to automate the bug detection process. Now, we run MemoryScape continuously, around the clock. It has been very effective in uncovering the hidden latent errors in our code. It finds problems that defy the regular testing methods, and it allows us to fix them proactively."

On the initial scan, MemoryScape found about 12 problems - and in every instance, it was a memory bug. MemoryScape has provided SIMULIA developers with a very effective, inexpensive way to find memory problems without spending a lot of time on the process. The data provided by MemoryScape's reporting mechanism provides details for developers to immediately see where a problem lies. When MemoryScape triggers an error, it saves a memory snapshot. This snapshot later gives a developer a very good picture of the place of the error, the memory contents, and the context of execution at that time.

The features of MemoryScape that SIMULIA developers have found to be the most valuable in their development process is the fact that the tool is scriptable and therefore can be run automatically. When the debugger finds a problem, it gives developers the information to see it right away and the fix is easy after that. This has led to tremendous cost and time savings at SIMULIA.

SIMULIA developers have also found MemoryScape to be faster in comparison to other checkers on the market that are much more troublesome to work with, are much slower, and require expensive instrumentation. "MemoryScape is fast enough that we use it to cover a lot of ground and the tool has provided us with a very efficient way of improving our QA substantially," Monyatovsky added.

Developers also like the feature of visualizing and analyzing memory leaks, and classifying them by source file. This information is invaluable, and is very hard to obtain by using anything else. Another major strength of MemoryScape is its

ability to execute and understand parallel MPI jobs, a key area of SIMULIA's focus right now. Having tools that operate in this environment is very important for the company's developers.

10 Future MemoryScape Product Plans

Areas for future development of MemoryScape include improved integration of the product into the TotalView Workbench Manager application. The Workbench provides a site for sharing configuration and session data between the different development tools that a developer may need. The Workbench provides an extensible base from which users can access recent debugging, memory debugging and performance analysis sessions.

Because integrated development environments (IDEs) are popular, but not universally embraced, the Workbench can be used from within an IDE but does not require the user to be using an IDE. Future versions of the MemoryScape product will more than likely share state and information about programs, input values and parameters, memory data files, etc. with the Workbench.

Other areas of future MemoryScape development include improved support for analyzing the historical usage of memory within the application so that developers can generate an accurate understanding of the memory usage behavior of the program. Perhaps the most frequently requested area of enhancement is in regards to providing developers with additional options for detecting and reporting memory reads and writes beyond the extent of heap allocations. TotalView Technologies is actively investigating possible enhancements that might allow developers to trade off runtime performance for more detailed information in this area.

11 Conclusion

Memory bugs can occur in any program that is being written, enhanced or maintained. These types of bugs are often a source of great frustration for developers because they can be introduced at any time and are caused by a number of different factors. They can also lurk in a code base for long periods of time and tend to manifest in several ways.

This makes memory debugging a challenging task, especially in parallel and distributed programs that include a significant amount of data and use a lot of memory. Commonly used development tools and techniques are not specifically designed to solve memory problems and can make the process of finding and fixing memory bugs an even more complex process.

MemoryScape is an easy-to-use memory debugging tool that helps developers identify and resolve memory bugs. MemoryScape's specialized features including the ability to compare memory statistics, look at heap status and detect memory leaks make it uniquely well-suited for debugging these parallel and distributed applications.

III
Performance Analysis Tools

Sequential Performance Analysis with Callgrind and KCachegrind

Josef Weidendorfer

Abstract This chapter presents the suite of tools Callgrind and KCachegrind. The first is an execution driven cache simulator, which outputs profile information on cache events, as well as the dynamic call graph of the execution, attributed with call counts and inclusive costs. KCachegrind is a visualization tool tailored at browsing the results gathered by Callgrind. After some introduction to sequential performance analysis and related tools, the tool suite is presented, followed by typical use cases. Finally, future developments are discussed.

1 Introduction

This chapter present the suite of tools Callgrind and KCachegrind [10], which mainly is used for sequential performance analysis. Development tools for program parallelization are the main objective of the "Parallel Tools Workshop". However, parallel code is composed of sequential code parts. Thus, the performance of frequently executed sequential code plays a significant role. One can classify performance bottlenecks of a parallel program into issues which also appear with sequential code, and issues only happening with parallel code, such as communication/synchronization overhead, or wasted time because of load imbalance. Both can influence each other. E.g., optimization of sequential code can change the load balance of parallel code. Sequential optimization can even require a more complex parallelization strategy. Further, when the partitioning of data in a parallel program leads to data partitions fitting into cache, this influences the sequential performance. Therefore, sequential and parallel performance optimizations are not independent from each other. As a rule of thumb, one should first go for the best sequential performance, e.g. by running the MPI program with one task only, and afterwards

Department of Informatics,
Technische Universität München, 85748 Garching b. München, Germany
Josef.Weidendorfer@in.tum.de

switch to parallel performance issues. In the following, when talking about performance analysis and optimization, only sequential performance is of concern.

First, a short overview of sequential performance analysis is given, presenting tools for sequential performance analysis. Then, the measurement tool Callgrind is presented, how it compares to the other tools, and its cache model and different features. After an overview of the visualization tool KCachegrind, typical usage scenarios are provided. The chapter concludes with features to be introduced in the future.

1.1 Short Overview to Sequential Performance Analysis

According to D. Knuth, "premature optimization is the root of all evil" [5]. Micro-optimization in the implementation phase is not only bad for code readability but it is also a waste of time for the developer. Optimization generally should be done after the implementation phase on bug-free code. It is important to concentrate on optimization of code parts where local performance improvements map to allover improvement. To find these code parts, performance analysis tools are used. While the localization of code to optimize is the main usage of such tools, it also is important to show what is going wrong, and to give hints about ways leading to performance improvements. Further usage scenarios for analysis tools are:

- Checking the correctness of assumptions on runtime behavior. E.g. when the tool outputs the exact number of times a function was called, and this differs much from the expectations of the programmer, there probably is a logical error somewhere. This is especially important when using library functions, and the time complexity of the library function differs from expectations.
- With the tool being able to measure single functions, one can directly decide about the best implementation from multiple alternative algorithms for a problem.
- Another use of analysis tools is to get knowledge about any unknown code. As the tool pinpoints the timely dominant code parts, one can assume that these are also the important parts of the code. By providing the call graph to these code positions, the ordering of how to get familiar with foreign code is given.

Reason of Bottlenecks in Sequential Code

After the tool pinpoints the code portions where most time is spent, and thus, where any performance optimization would show best allover improvement, the actual optimization depends on the type of bottleneck. The reason of bottlenecks in sequential code can be categorized as follows:

- Logical errors that do not influence the correctness of the program but have negative impact on performance. This includes redundant calls to functions with

idempotent results such as multiple initialization, or function calls done always but needed only sometimes depending on input. If the tools shows that time-dominant functions are called excessively often, or loop counts inside of the function are unexpected high, a logical error could exist here. To become aware of such an issue, the tool needs to collect call and jump/loop counts.
- Wrong algorithm for a problem with unneeded high runtime complexity. The solution is to test different algorithms.
- Bad runtime behavior dependent on system architecture characteristics, resulting in slow code execution. Reasons for execution stalls are (1) memory accesses missing the cache, thus waiting for data from slow main memory, (2) unpredictable control flow changes, (3) data dependencies limiting the instruction level parallelism, (4) further issues depending on microarchitectural limitations. With modern processors, bad memory access behavior is by far the biggest problem, as a cache miss can last hundreds of processor cycles. The tool needs to be able to show the exploitation of processor resources. This usually needs hardware to support the collection of according event types, and the tool being able to access this hardware support.

In the scope of high performance computing, it is worth to avoid bottlenecks in every category, as even minimal relative runtime improvement can map to significant absolute improvement, taking into account the typical long runtimes. While the first two types of reasons for bottlenecks are usually taken care of after the implementation phase of a program, the last category needs further analysis every time the hardware changes. Especially for architecture aware optimizations, it is important to understand why a given code property results in stalls at some points in the microarchitecture in order to be able to find fitting solutions. While it can be expected that improvements in cache exploitation lead to real performance improvements, this is less the case for the other reasons for pipeline stalls mentioned above. However, every optimization step should be checked for real time improvements. A reduction of some event count measured by the tool is not enough.

Performance Measurement Techniques

A tool for sequential performance measurements typically allows to measure event types such as clock ticks (ie. time), function calls, percentage of bus utilization, or cache misses. These events have to be related to the code region where events happen, or even better, also to the full call path starting from `main` down to the code region. This allows the developer to identify the context of event occurrences more easily, especially when a function is called from different places, or the functions is inside of a library unaccessible to the programmer. For the latter, any code changes would have to be done up the call chain.

Storing the occurrence of every single event is often not possible because of the high amount of data this would produce. As the tool usually runs on the same hardware as the program to be measured, resource consumption of the tool itself

should be kept to a minimum to not influence the measurement, thus destroying the usefulness of the measurement itself.

There are different solutions for minimizing the impact of a performance analysis tool:

- Online aggregation. Instead of storing a sequential stream of time-stamped events (a *Event Trace*), counters are incremented. Multiple counters can be used for different code positions, resulting in a histogram of event counts. This is called a *flat profile*. Summing up the counts for each function gives *exclusive* costs, ie. event counts for events happening in this function. By also relating event occurrences to all the functions up in the call path (*call path profiling*) gives *inclusive* costs for functions, ie. event counts for events not only happening in a given function, but also in all functions called from there. The advantage of online aggregation is that the size of the measurement data depends on code size, and not runtime (as is true for full event traces).
- Statistics instead of exact counts. The distribution of events to code positions typically does not change much when only every *n*-th event is checked. This is called *Sampling*, and is supported by all processors nowadays with hardware performance counters. A counter for a specific event type can be configured to trigger an interrupt calling into the tool, after a given number of events occurred. The advantage is that the overhead of the measurement tool is tunable, and there does not need to be any *instrumentation* of the target binary for the tool to work. Instrumentations are code modifications needed for basic functionality of a analysis tool. Any such instrumentation is an overhead potentially disturbing the measurement.
- Architecture simulation. This systematically avoids any influence of the measurement tool on the measurement. However, for practical reasons, the simulation slowdown has to be acceptable. This usually means that the model has to be quite simple, and only part of the microarchitecture of a processor is covered.

1.2 Related Tools

The best known tool for performance analysis of sequential code probably is GProf [3]. It uses Sampling based on time intervals (available in every OS) and compiler instrumentation, to get the exact count of method calls. The latter allows to heuristically build up the call graph and calculate inclusive costs. However, for this, the application as well as every shared library used needs to be recompiled with a special compiler flag for instrumentation for correct results. Unfortunately, the instrumentation often leads to high measurement overhead, especially when tiny functions are called often. Additionally, the heuristic for the calculation of inclusive costs can go wrong.

With the availability of hardware performance counters in processors, the Sampling method is most commonly used today by tools from hardware vendors such as Intel, Sun, SGI and others. Intel VTune [4], which is available for Intel processors,

running on Windows or Linux, allows sampling both system-wide and per process. It also has a mode for collection of the call graph using binary instrumentation. OProfile [6], available for almost any architecture running Linux, does system-wide profiling with the need for root access, taking advantage of performance counters. Sun Performance Analyzer [8]) is similar to VTune, running both on Linux and Solaris as part of the Sun Studio IDE. Vendor tools usually offer sophisticated binary instrumentation to get the call graph and exact call counts. However, this can have the same overhead issues as GProf.

For sure, architecture simulation is an important tool in the design process for every hardware vendor, but these in-house simulators are not publicly available. Besides, cycle-accurate simulators probably are not practically useful for performance optimization because of their huge slowdown. However, there is a system simulator available from AMD with are very simple CPU model [1]. Further, there were quite some simulators developed for architecture research such as [2, 9]. They typically rely on offline memory traces, as they are tailored for parameter studies. Callgrind, presented in the next section, specializes in ease-of-use for the purpose of performance analysis, helping programmers to optimize their code.

2 Callgrind: a Call-Graph building Online Cache Simulator

Callgrind is a performance analysis tool based on architecture simulation. The simulation is execution driven and is done together with event aggregation on the fly, ie. simultaneously to the execution of the target code. Calling into simulation needs instrumentation, which is done at runtime, thus allowing the tool to work on unmodified compiled code.

The dynamic runtime instrumentation is provided by the open-source package Valgrind[1] [7]. Valgrind includes a set of tools for correctness checking and performance profiling, such as an memory correctness checker (Memcheck), a race detector (Helgrind), a memory profiler (Massif), and performance analysis tools based on cache simulation (Cachegrind, Callgrind). With its runtime instrumentation infrastructure, Valgrind Tools can observe user-level processes compiled for Linux or AIX on x86, x86-64, PPC32, or PPC64. Valgrind, as well as Callgrind, is open-source covered by the GPL.

Callgrind is more or less an extension of Cachegrind, optionally using its cache simulation model, but adding the ability to track any calls happening in a program run. While Cachegrind provides a flat profile of the number of cache events happening in functions (ie. *exclusive* costs) in its output, Callgrind also provides *inclusive* costs with the help of call tracking. In contrast to GProf, which heuristically calculates the inclusive cost from call counts, Callgrind directly collects it by storing the value of a global event counter at function enter, and subtracting it from the counter value at function exit.

[1] Valgrind homepage: `http://www.valgrind.org`

Together with a graphical visualization of the call graph, this allows to see the cost distribution starting from `main()`, and going down the call chain to the function where most cost is spent. When these costs are spent deep down in some 3rd-party library, it is easy to recognize the own code which is responsible for calling the library, and this exactly is the position where changes are required for improvement.

2.1 Cache Model and Events

The cache simulator models a synchronous, two-level, inclusive cache with separate L1 instruction and data caches, and an unified L2 cache. *Synchronous* means that the simulator always handles cache accesses at once, and there can not be multiple access requests simultaneously in completion. An *inclusive* cache hierarchy always fully contains the cache lines of smaller levels in higher, larger cache levels. Writes always are passed from the L1 (ie. L1 is write-through) to the L2 cache, so that afterwards, the written-to cache line occupies space both in the L1 data cache and the L2 unified cache. Both caches work with user-level addresses (ie. virtual addresses); there is not simulation of a translation lookaside buffer (TLB). At every memory access, the simulator first checks the level-1 cache (depending on the access either the L1 instruction or L1 data cache) for the cache-line holding the accessed address, and on a miss, it also checks the level-2 unified cache. Whenever there was a miss in L1 or L2, space for the according cache line is reserved, possibly evicting another line. The replacement strategy used is LRU.

This cache model resembles a simplification of the cache hierarchy used e.g. in Intel Pentium-3/4/M single-core processors[2]. Callgrind (and Cachegrind) by default check the real processor[3] for L1/L2 cache sizes, cache line length, and associativity. The idea is that the user probably wants the simulation to be similar to the reality on the processor of the machine the simulation runs on. However, these cache parameters can be explicitly given on the command line, too.

Events generated by the cache simulation are L1 hit, L2 hit, or L2 miss. For each of these three results, the type of access (instruction read, data read, or data write) is noted, too, so that there are nine different possible event types. In the output, counters for these events are given per source line, or optionally even per instruction address. The set of event types does not specify the kind of eviction triggered by a miss. For a L2 write-back cache, the dirtiness of a cache-line (ie. modified or not) would have influence on the cost (bus occupation), as a modified line needs to be written back. However, because the events do not distinguish between different miss kinds, the cache model thereby does not specify whether the L2 cache is write-back or write-through: both cache types result in the same event counts.

[2] Newer Intel processors using the Core microarchitecture have a write-back L1 cache, leading the a slightly different behavior. AMD processors always had *exclusive caches*, where on a miss, data is always loaded into L1, and lines evicted from L1 are stored into L2 (acting as *victim cache*), leading to different content in L1 and L2.

[3] via the CPUID instruction

It is important to note that the cache simulation can not say anything about the stall time in the processor core because of cache misses. This would need an cycle-accurate simulation not only for the cache, but also for the CPU microarchitecture, and probably more important, for the system bus, memory controller, and DRAM chips. Aside from the fact that hardware documentation for a given processor is not available in the detail needed, the simulation would not by practically useful any more because of the simulation slowdown. However, as memory accesses often slow down application performance in practice, a relative reduction of L2 misses often maps to faster runtime performance. Quantitatively, one can construct a heuristic giving worst-case cache latency by measuring the worst-case miss latency of an L1 and L2 miss on a real machine, e.g. with the Calibrator tool[4], and use this as cycle estimation. This worst-case *derived*[5] event is provided as "cycle estimation" event (CEst) within KCachegrind. For adjustment to the cache latencies of a given processor, the coefficients in the formula of this derived event can be edited. However, this worst-case heuristic can be wrong because of other application activity (such as lots of heavy calculations), partially or even completely hiding the cache latency.

Comparison with Real Processor Caches

The following discrepancies of this simple cache model to any real processor (best comparison would be e.g. an Intel Pentium-M) can be noted:

- Synchronousness. Any real cache hierarchy can have multiple requests in the fly, handling them in an asynchronous way. While this would drastically raise the complexity of the simulator, the event counts get far more difficult to interpret and understand: E.g. 10 accesses in a row to the same not-loaded cache line lead to up to 10 L1 misses and only one L2 miss for the asynchronous case, depending on the timing of the accesses and on the out-of-order capability of the execution pipeline.
- No simulation of *hardware prefetchers*. Every cache implementation nowadays includes automatic prefetching of data by predicting future accesses. Thus, real accesses potentially find the needed data in the cache, depending on the correctness of the prediction and the pre-loading in time, which also depends on the bus load. The influence on application performance can be significant. However, the existence of HW prefetchers is architecture dependent. Optimization based on the simple cache model will result in faster performance also on processors without prefetchers.
- No difference in events between write-through/write back L2. Cache hierarchies in all processors have a write-back behavior on the last level to reduce bus activity. Thus, writing back dirty evicted cache lines can have a performance impact. However, this happens on writes, and typically, a write transaction can be done

[4] http://monetdb.cwi.nl/Calibrator

[5] A derived event is an event not measured directly, but calculated from other events

completely in the background, not influencing the runtime (in contrast to a memory load, where the value is needed for further processing).

While these differences to real processor hardware exist, reduction of cache events of the simple model usually also lead to reductions of events on real hardware. And typically, it also results in reduction of runtime. However, as already stated in the introduction to this chapter, additional runtime measurements are always needed to check that there is real runtime improvement.

There are important benefits to a simple cache model aside from the simulation time:

- The results of the simple cache model are easy to understand and, with help of event attribution on instruction level, also easy to reconstruct. They usually match an a-priori analytical analysis of the cache behavior of the code. This allows to better estimate the improvement of code caches beforehand, and therefore, leads faster to satisfying optimization results.
- The cache simulation results are reproducible. For comparison of the effectiveness of a code optimization, it is far better to be able to rely on stable and reproducible measurement results. Results on real processors can vary heavily with the same code from execution to execution, depending on history and other background system activity. This usually leads to the need for averaging runtime results of time consuming, multiple runs.
- Is is good when optimizations work on any architecture. So-called *cache oblivious* algorithms exploit caches whatever the capacity is. Optimizations working in a simpler cache model typically lead to better architecture independence of the runtime improvement. On the other hand, it can happen that no improvement can be observed with sufficiently sophisticated hardware.

Extensions to the Cache Model

The basic cache model of Callgrind, as described in the previous section, is the same as found in Cachegrind. In Callgrind, there are further options to extend this simple cache model in two ways:

- Explicit specification of write-back behavior for the L2. Each of the three L2 miss events (for instruction read, data read, data write) are further subdivided into a L2 miss event with dirty and non-dirty state, respectively. For the cycle estimation derived event using worst-case cache latencies, the formula can be extended to account the double time for the three additionally added L2 miss events evicting a dirty, modified line before. The motivation is that there are two bus transactions instead of one[6]. While the cache extension seems to be useful, the results usually

[6] Unfortunately, the previously mentioned Calibrator tool does not measure L2 misses with dirty line eviction.

do not differ much from the basic cache model[7]. This write-back extension is switched on with the command line option "–simulate-wb=yes".

- Addition of a best-case hardware prefetcher. As stated in the introduction, every processor nowadays has mechanisms which try to predict future memory accesses and prefetch data which is expected to be accessed in the future by a program. In the optimal case, a prefetch is completed when the data is needed, thus fully hided memory latency. The prefetch simulation predictor detects at most 16 sequential stream accesses (up or down) inside of memory pages of 4 kB when these are the only accesses in a page. On an memory access which is part of a detected stream, the cache line which is 4 lines apart from the given access in the detected direction is loaded into cache without any latency at all. Thus, it is assumed that every prefetch can be fully hidden (ie. best-case). This simple stream prefetcher scheme should be part of any real hardware prefetcher implemented in processors nowadays. For example, it is quite similar to the one found in recent Intel processors for the L2 cache. When to only option to optimize for memory accesses is the insertion of software prefetch instructions, this cache model extension can guide where such prefetch instructions are really needed, and where they are unneeded, as covered by the hardware prefetcher. This is important as the insertion of software prefetch instructions also can slow down code because of limited instruction decoding bandwidth. The prefetcher extension is switched on with the command line option "–simulate-hwpref=yes".

Metrics Extension: Cache Exploitation

While not strictly part of the cache model, an important property of a simulator are the event types and metrics which are derived from the simulator state changes. The default for Callgrind are, as mentioned above, nine cache events, providing the result of accesses into the cache hierarchy as L1 hit, L2 hit or L2 miss. These are the events which typically also can be measured with hardware performance counters, and hint at code positions where there is a potentially stall because of slow memory accesses. However, an question not answered by these events is, how the cache is exploited by the application. For any cache optimization, the goal is to improve the *locality* of memory accesses. This can be subdivided into *temporal* locality (the same memory cell is accessed multiple times) and *spatial* locality (memory accesses are to nearside memory cells). Processor caches are beneficial because these two types of locality generally appears in any program, which is an observation called *Principle of Locality*. Holding copies of memory cells exploits temporal locality, and caching blocks of memory does exploit spatial locality.

To show potential for better cache exploitation, it is important to quantify the two locality types. The needed information can be captured via the cache simulator

[7] There is an interesting exemption: Searching for prime numbers using the algorithm "Sieve of Eratosthenes" has a lot of unnecessary writes where the latency can not be hidden by processors. By getting rid of them, real caches show doubled performance. To show this effect, this write-back simulator extensions has to be switched on.

state. For temporal locality, it is important to know how many times a given cache lines was accessed before it was evicted, and for spatial locality, the percentage of bytes really accessed in a cache lines before being evicted is relevant.

Any visualization of program performance should point at bad behavior. Therefore, the new event types introduced for cache exploitation should be large for bad locality behavior. The following events are generated via the command line option "–cacheuse=yes":

- *Access Cost* events `AcCost1` for L1 and `AcCost2` for L2, respectively. These events show bad temporal locality. As KCachegrind/Callgrind currently only can handle integer values for event types, a value of 1000 is defined as Access Cost for a cache miss. When there are two accesses to a cache line before eviction, the cost is 500 for both of the accesses, and so on. Because of practical issues, this model has to be changed: we need to relate the event to a code position. The solution is to relate the access cost of all accesses to a cache line to the code position which triggered the load of the line.
- *Spatial Loss* events `SpLoss1` for L1 and `SpLoss2` for L2, respectively. These events show bad spatial locality by providing the number of bytes not accessed by the processor before eviction. Thus, `SpLoss2` directly gives the size of data which was unnecessarily loaded into L2. This can be set into relation to the full amount of data loaded into L2, which is the number of L2 misses multiplied by the cache line size. Regarding relation of this event to a code position, the unneeded bytes of a cache line are attributed to the position which triggered the cache line load.

The used code relation for cache exploitation events unfortunately is not easy to interpret. It would be better to use attribution to data structures, which currently is not supported in Callgrind and can not be visualized in KCachegrind. At least, from the code position, it can be looked up which data was accessed. With bad spatial locality, the layout of the data in memory should be redone. It generally is better for the cache to put data nearside which is used in a similar way by the program. Only with bad temporal locality, code restructuring is needed.

2.2 Additional Callgrind Features

Callgrind consists of two main features: (1) the tracking of calls for building up the call graph, getting call counts, and enabling collection of inclusive cost, and (2) the cache simulator, which already was described in length. The first feature is useful on its own, and as the cache simulator adds to the slowdown, it is switched off by default. Without any options, only number of executed instructions executed (`Ir` for "instructions read from cache", which gives the same value in the simple cache model) and function calls are collected.

2.2.1 Control Flow Collection

When analyzing a function alone, the control flow (e.g. number of loop iterations) is interesting. This needs the collection of (conditional) jumps executed, and is switched on via command line "–collect-jumps". The jumps can be visualized in KCachegrind next to the source code. However, there is an issue: as performance of the compiler optimized version has to be analyzed, there can be quite some restructuring done between source code flow and assembly code, leading the strange jump annotation in the source. To avoid this confusion, it is better to look at the assembly code annotated with jumps. For Callgrind to give output at instruction level, one has to use the option "–dump-instr=yes", which should be used always when analyzing the control flow.

In KCachegrind, unconditional jumps are shown in blue, whereas conditional jumps are shown in red (see Fig. 4).

Avoiding Slowdown for Uninteresting Program Phases

Often, there is uninteresting initialization going on at program start. Such program phases can be executed with a slowdown of only factor 2 – 3 (which is the slowdown of running a code in Valgrind without any additional instrumentation) by changing the instrumentation mode to "off". For the instrumentation mode to be off at program start, use "–instr-atstart=yes". For toggling instrumentation mode when entering/leaving a function, use "–toggle-collect=*function*". Note that *function* is allowed to contain wildcards "*" (matching zero or more arbitrary characters) and "?" (matching one arbitrary character).

Instrumentation mode also can be explicitly set in the program to be analyzed by including the header `callgrind.h` and using a *Valgrind Client Request* e.g. by inserting the C preprocessor macro `CALLGRIND_START_INSTRUMENTATION`. For further details, see the Callgrind online manual.

When cache simulation is done, it is important to understand that the cache is fully flushed at changes of the instrumentation mode. To not see the many cold misses, one should use a "warm-up" phase after switching on instrumentation, where no collection of events is done. For details on switching on/off the collection mode, see the online manual.

Interactive Control

While Callgrind is running, its current status (function call stack, events collected) can be requested any time with "callgrind_control -b". There are more interactive commands, like requesting a profile dump with option "-d", switching on/off instrumentation mode (see last section), and so on. See the manual for this command for details.

While not really interactive, the same commands can be triggered by the program being observed, either via client requests (similar to the example use of the macro `CALLGRIND_START_INSTRUMENTATION` above), or via specifying the triggering code position on the command line. E.g. "–dump-before=*function*" generates a new profile dump every time the given function is entered.

Cycle Avoidance

In issue of general concern to profiling tools is how to handle mutual recursion[8]. Recursion happens when one or multiple functions calling each other in a mutual way. Then, the form a so-called *cycle*. This can be problematic for program analysis because inclusive cost is not defined for calls inside of a function cycle. Despite of this, Callgrind collects bogus "inclusive cost". The visualization tool has to detect function cycles and prohibit the visualization of inclusive cost inside of them.

For small cycles, this is no issue for the analysis. For large cycles covering much of a programs functions, the benefit of collecting inclusive cost is lost. The importance of inclusive cost is highlighted by the fact that it is used to cut off the uninteresting portions of the call graph visualization. Thus, with large function cycles, the call graph visualization often is useless, too.

To understand the full problem, one has to realize that from a profile dump which only gives the number of calls happening, it can not be detected whether existing calls really happened in a recursion, given that they can be ordered in a way to produce a function cycle. E.g. with calls $A \to B$, $B \to C$ and $C \to A$, one can construct the recursive cycle $A \to B \to C \to A$. The visualization has to assume that this cycle really happened, and can not provide inclusive cost any more. This is true even if there really only existed the call chains $X_1 \to A \to B \to Y_1$, $X_2 \to B \to C \to Y_2$, and $X_3 \to C \to A \to Y_3$ (the assumed recursion here is called a *false cycle*). In the paper on GProf [3], there is a detailed explanation of how profile visualization tools handle this issue (KCachegrind does the same). In effect, artificial functions called like "cycle1" are introduced for potential recursions, consisting of all the functions which are part of recursion.

For programs using event based program style (e.g. GUI code), which are mapped to callback mechanisms by the compiler, there often are cycles covering most of the functions of the program. For this, Callgrind has options which can avoid these functions. One such solution is to store not only the code position of a event, but a call chain as context. For further details, see the Callgrind manual chapter about Cycle Avoidance.

[8] This seems to be less important for HPC code, which is the main target in this Workshop.

Miscellaneous

While mainly targeting sequential performance analysis, Callgrind can produce different profile output for each thread of a multi-threaded program. This mode is switched on with "–separate-threads=yes".

3 KCachegrind: Profile Visualization

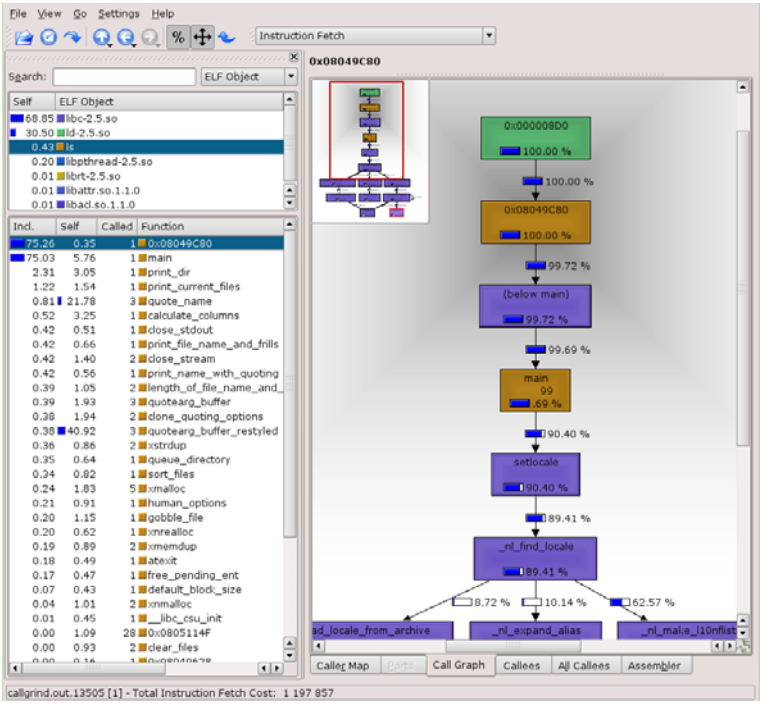


Fig. 1 A screenshot of the KCachegrind window, showing the ordered function list on the right and the call graph visualization around the active function (center of the dark background shading) on the left

While there is a command line tool for providing sorted function lists as ASCII output on the profile data ("callgrind_annotate"), and which even can print annotated source code, it is only some kind of work-around for people not able to have KDE libraries installed on their machines. The command line tool is missing cycle detection and handling (see 2.2.1), as well as assembly code and jump annotations.

Profile data files generated by Callgrind is stored in the same directory where Callgrind was started. The file name has the form "callgrind.out.*PID*" (possibly

appended by further numbers when multiple dumps are generated). Here, *PID* is the process ID of the observed run[9], and this number also is printed as prefix in the log output produced by Callgrind on the terminal (see 4). When there is only one profile data file in the current directory, simply starting KCachegrind will pick it up. With multiple files, one can provide the file to load on the command line, or use the "File/Load" dialog.

Fig. 1 shows the general layout of the KCachegrind GUI. For all visual parts of the window, there exists a "What's this" help text. It is shown on selecting a window part after pressing Shift-F1.

3.1 Basic concepts

There are a few important concepts. The basic item visualized by KCachegrind is a *Cost Item*, which can be a function group (ELF object, source file, C++ class, function cycle), or a function itself. There is always one cost item *active*, and this item can be chosen, and is shown selected in the function/group list on the left of the window (activating a function group is done by enabling the group type via the combo on top of the left list, and double-clicking on the group itself; to active a function, a single click in the function list is enough). The active cost item is visualized on the right side of the window, in different visualization views (split in top and bottom areas; views can be moved between areas via a popup menu appearing by right clicking a view tab title). The name of the active item can be seen on top of the visualization views. Every visualization is done around, and based on the active item. See below for a description of the different view types. Inside of visualization views, there also exists the possibility to choose a cost item with a single click which becomes *selected*. While a change of the active item changes all the visualizations, centering them around the active item, a change of the selected item does only change the highlighting of this item in all existing visualizations. This allows to have a synchronized, highlighted view to the selected item in all visualization views. Double-clicking on any item in a visualization view changes the active item. Nearside any function name, there is a small colored rectangle. When function grouping is switched off, this simply is a color coding for this function, derived from the function name itself. When function grouping is on, the color identifies the function group, and not the function itself. This color, identifying either the function group or the function, is also used for coloring the visual representations of the functions (or the group they are part of) in different visualization views, thus allowing the better visual identification of the appearance of the identical function.

Callgrind eventually provides multiple values for different event types attributed to each cost item. Similar to the active item, there exists an active *Event Type*[10]

[9] The observed executable runs in the same process as Callgrind. This is a generic property of Valgrind tools.

[10] In addition, there optionally is a second active event type, which is useful to compare two event type values in visualization views.

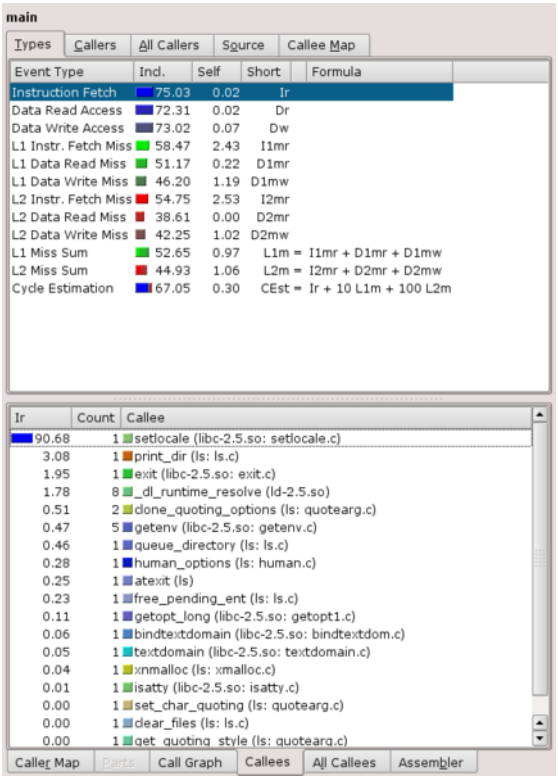


Fig. 2 Visualization panes showing the list of event types collected (top), as well as a list of callers/callees of the selected function (bottom)

specifies the event type used for all the numbers shown near to cost items both on the list on the left as well on visualization views on the right. For each event type, there can be exclusive (also called *Self Cost*) and inclusive cost (described in 2), depending on the cost item type: inclusive cost is only defined for functions (and also used as attribution of calls). For function groups, or subitems such as source/assembly lines, only exclusive cost is provided. Aside each cost value, there is a small colored bar shown. It provides a fast visual feedback whether the cost is important or not. The color actually encodes the event type: blue color shades means instruction/data reads/writes hitting the L1, thus allowing quite full CPU utilization. Green corresponds to L2 hits, and red to L2 misses. See the top of Fig. 2 for the exact color coding. For derived events, the colors are mixed depending on the percentage a given basic (ie. measured) event type attributes to the value of the derived event. This is given by the coefficients in the formulas of derived events. This makes it easy e.g. for the "Cycle Estimation" event to see the partitioning on estimated time spent in computation (blue), in waiting on L1 misses/L2 hits (green) and L2 misses (red).

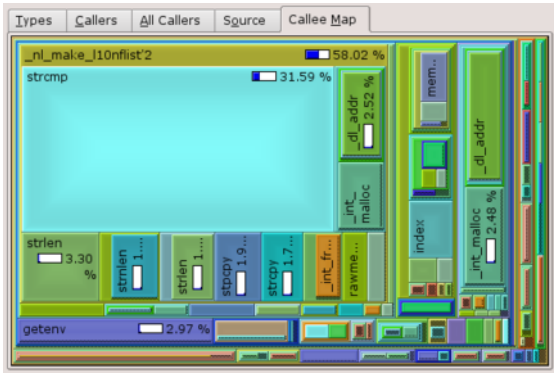


Fig. 3 Visualization panes showing the Callee Map

Description of Window Parts

At the top of the window there is a menu bar, with available items mostly self-explaining. Below, often used actions are provided in the tool bar. Not obvious is the third button from the left, which requests a profile dump from a simultaneously running Callgrind in the same working directory, and reloads all profile portions afterwards. It provides the only interactive control to Callgrind from KCachegrind. The button with the arrow cross changes the relation of percentage values. When pressed, 100% always relates to the cost of a parent item: e.g. source line cost to function cost, or cost attribution in the call graph to the selected function. When not pressed, the relation is always to the total count of the currently selected event type. On the right of the arrow button, cycle detection can be switched off (mostly interesting for analysis of GUI applications). Finally, on the far right, there is a combo box allowing to change the active event type (see basic concepts above).

On default, function grouping is switched off on the left side of the KCachegrind window. This side shows a list of functions, sorted in descending direction either by inclusive cost, exclusive cost, call count in lexical order of the function or function location. The actual ordering can be chosen by clicking an the according header cells. This list shows only the most important functions according to the selected ordering, to make the GUI fast even with profiles with hundred thousands of functions. To search a given function, start typing its name in the search box. The search is incremental, and results will update on each key press. Next to the search box, there is a combo box where the function grouping type can be selected. On any grouping other than "None", there will be an additional smaller list shown with available function groupings of the selected type. The function list itself only shows functions belonging the the selected group in this mode.

Most important is the window part with the visualization views on the right. In Fig. 1, the *Call Graph* visualization is shown. It shows a visualization of only a small part of the full call graph of the program, centered around the active function. The active function is visually emphasized as the center of a dark background shading, enabling the active function to quickly be recognized also in the birds-eye graph

overview in the top left. The threshold for the cutoff of the shown call graph part can be changed in the context popup menu, which is shown on right click into the view.It is clear that the cost of callees downwards is always less than the cost of the active function. However, this is also true in caller direction upwards. To understand this, one has to note that only that part of the actual cost of any caller/callee is shown, that is also spent in the active function itself. Thus, if there are 2 callers of the active function, one can see the distribution of time coming from one two call chain, respectively. Also, if e.g. the `main` function is shown as caller in the call graph, its cost still can not be more than the cost of the active item. The *selected* item can be changes by clicking on it, or by keyboard navigation. Note that also calls can be selected. As mentioned above, when there are multiple visualizations visible, a change of the selected function will highlight this function in every visualization.

Quite simple visualization views are the *Caller/Callee* lists. As example, at the bottom of Fig. 2 the Callee visualization is shown. There exist extensions to these simple views titled *All Callers/Callees*. There difference is that these not only show the direct callers/callees, but all callers/callees reachable from the active function. At the top of Fig. 2, the event type visualization is shown, which is a list of all event types with according values for the active cost item. Clicking on an event type line changes the active event type. Further, new derived event types can be added here via the context menu, and formulas of derived event types can be edited.

A more sophisticated visualization view is the *Callee Map*, shown in Fig. 3. In contrast to the call graph view, it shows the call relationship as nesting of rectangles. Callees are drawn inside of the caller rectangle[11]. The area size of a rectangle is proportional to the inclusive cost of the function this rectangle represents. Thus, it is easy to spot functions with large inclusive costs, even if there are a lot of callees, or when the indirectly called function is a large distance away in the call graph.

Further, Fig. 4 shows the source and assembly annotation views, here additionally with jump visualization (red are conditional jumps, blue are unconditional ones). On lines where calls/jumps are starting, for every call there is a further line inserted into the source/assembly, showing call/jump counts. Selecting these lines will select the corresponding call/jump as *selected* item. It is interesting to note that the last column in the assembly annotation list shows the debug information the compiler produced for the given instruction, ie. the related source line number.

Not shown here is the *profile part* visualization. When there are multiple profile parts are loaded into one window (via the File/Add... item), this *profile part* list becomes available. Here, the set of active profile parts can be chosen whose costs are to be shown. For any other parts, as the optional side bars, see the available "What's this" help.

[11] The Callee Map visualization is a so-called *Tree Map*, actually showing call trees and not call graphs. Thus, selecting a function in another view can highlight multiple rectangles, which all representatives of the selected function, but part of a different call chain.

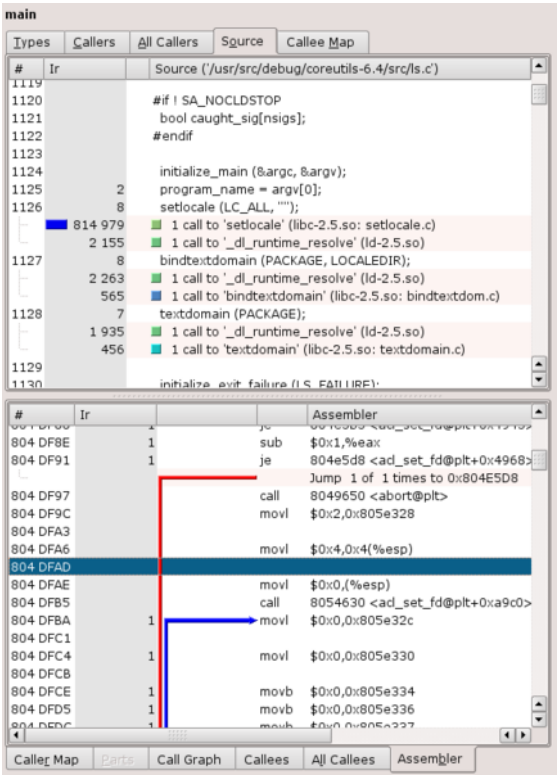


Fig. 4 Visualization panes showing the source (top) / assembly (bottom) annotation views, in the bottom additionally with jump annotation

4 Usage Example

In the following, we show the influence on cache behavior of iterating over a matrix either row- or columnwise. To example code is shown in Fig. 5. After compilation with `cc matrix.c -o matrix`, a Callgrind run without further options (ie. without cache simulation), produces the following output:

```
weidendo@lapbode:~tmp> valgrind --tool=callgrind ./matrix
==18766== Callgrind, a call-graph generating cache profiler.
...
==18766== For more details, rerun with: -v
==18766==
==18766== For interactive control, run 'callgrind_control -h'.
==18766==
==18766== Events    : Ir
==18766== Collected : 31122717
==18766==
==18766== I   refs:      31,122,717
```

When visualizing the resulting profile in KCachegrind, one can see that each of the two matrix sum functions has exactly the some number of instruction executions (absolute values probably will be different depending on architecture, com-

piler, compiler version, version of C runtime library and so on). On the authors machine these are 11,007,012 instructions.

A more elaborated Callgrind execution with cache simulation on and profile dump information and instruction level, including jump collection, is given by

```
weidendo@lapbode:~tmp> valgrind --tool=callgrind \
           --simulate-cache=yes --dump-instr=yes \
           --collect-jumps=yes ./matrix
==19136== Callgrind, a call-graph generating cache profiler.
...
Events    : Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
Collected : 31122763 13042032 6021808 569 1125785 125317..

I   refs:      31,122,763
I1  misses:           569
L2i misses:           564
I1  miss rate:        0.0%
L2i miss rate:        0.0%

D   refs:      19,063,840  (13,042,032 rd + 6,021,808 wr)
D1  misses:     1,251,102  ( 1,125,785 rd +   125,317 wr)
L2d misses:       370,406  (   245,101 rd +   125,305 wr)
D1  miss rate:        6.5% (       8.6%   +       2.0%  )
L2d miss rate:        1.9% (       1.8%   +       2.0%  )

L2 refs:        1,251,671  ( 1,126,354 rd +   125,317 wr)
L2 misses:        370,970  (   245,665 rd +   125,305 wr)
L2 miss rate:         0.7% (       0.5%   +       2.0%  )
```

At the end of the run, the event types together with the total counts are shown, and afterwards some statistic about cache miss rates. The shown percentages always relate to number of instruction reads, data reads, and data writes issues by the processor (Ir/Dr/Dw). Therefore, "L2 rate" figures provide the rates of the full 2-level hierarchy.

Using KCachegrind, it is interesting to select the derived "Cycle Estimation", and to compare the color distribution of the bars next to `rowwise` and `columnwise` in the function list on the left. This gives an idea of how the time is spent on computing phases (blue), on waiting for L1 (green) and waiting for L2 (red). On can see that while `columnwise` shows a lot of L1 misses, L2 misses actually are lower than for `rowwise`. Depending on cache size and latencies, the best function seems to depend on the architecture. However, when adding the hardware prefetcher with "–simulate-hwpref=yes", the real advantage of `rowwise` becomes visible: all accesses can be prefetched (remember this is a best-case estimation).

5 Future Development

The cache simulator in Callgrind currently is extended to allow multi-core simulation, with as many cache hierarchies as there are threads in a multi-threaded pro-

```
double A[1000][1000];

double rowwise() {
  int i, j;
  double sum = 0.0;

  for(i=0;i<1000;i++)
    for(j=0;j<1000;j++)
      sum += A[i][j];
  return sum;
}

double columnwise() {
  int i, j;
  double sum = 0.0;

  for(i=0;i<1000;i++)
    for(j=0;j<1000;j++)
      sum += A[j][i];
  return sum;
}

int main() {
  int i, j; double sum1, sum2;

  for(i=0;i<1000;i++) for(j=0;j<1000;j++) A[i][j]=1.0;

  sum1 = rowwise();
  sum2 = columnwise();
  return (sum1 == sum2);
}
```

Fig. 5 Example code for Callgrind usage

gram. The main purpose is to see how the working set of the different threads relate to each other. This knowledge could be used for core/thread bindings best exploiting available resources. In additions, coherence issues like false sharing could be detected. For KCachegrind, this needs adequate visualization.

For the Callgrind profile format, optional inclusion of assembly and source code information is planned. Together with command line tools adding this information, KCachegrind will not need to be run any longer on the same architecture (for producing disassembly code / loading source code), and visualization of multiple code versions (before/after optimization) will be easier.

For KCachegrind, quite some improvements are planned since long ago, which we hope to implement soon. This includes issues such as providing a command line version for simple merging/filtering of profile data, as well as a Qt-only version in addition to the existing KDE version. From the functionality side, handling of more complex formulas for derived events is planned as well as a comparison mode for different measurements.

Acknowledgements Thanks to Julian Seward for the excellent runtime instrumentation framework Valgrind, and to Nicolas Nethercote for Cachegrind, which Callgrind is based on.

References

1. AMD: AMD SimNow Simulator. http://developer.amd.com/tools/simnow/Pages/default.aspx
2. DeRose, L., Ekanadham, K., Hollingsworth, J.K., Sbaraglia, S.: SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In: Proceedings of SC 2002. Baltimore, MD (2002)
3. Graham, S., Kessler, P., McKusick, M.: GProf: A Call Graph Execution Profiler. In: SIGPLAN: Symposium on Compiler Construction, pp. 120–126 (1982)
4. Intel: Intel VTune Performance Analyzer. http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm
5. Knuth, D.: Structured Programming with go to Statements. ACM Journal Computing Surveys **6**(4), 268 (1974)
6. Levon, J.: OProfile, a system-wide profiler for Linux systems. http://oprofile.sourceforge.net
7. Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007). San Diego, California, USA (2007)
8. Sun: Sun Studio 11: Performance Analyzer, Reference Manual (2005). http://docs.sun.com/app/docs/doc/819-3687
9. Tao, J., Schulz, M., Karl, W.: A Simulation Tool for Evaluating Shared Memory Systems. In: Proceedings of the 36th ACM Annual Simulation Symposium, pp. 335–342. Orlando, Florida (2003)
10. Weidendorfer, J., Kowarschik, M., Trinitis, C.: A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In: ICCS 2004: 4th International Conference on Computational Science, *LNCS*, vol. 3038, pp. 440–447. Springer (2004)

Improving Cache Utilization Using Acumem VPE

Erik Hagersten, Mats Nilsson and Magnus Vesterlund

Abstract The move to multicore offers a steep increase in compute power, while little is done to improve the performance of the memory system. Typically, current applications make poor use of the memory system and few developers have the insight to fix such problems. Furthermore, the introduction of shared memory system resources makes the picture even more complicated.

Acumem Virtual Performance Expert (VPE) automatically identifies wasteful memory access behavior in applications and suggests improvements. About 20 different types of performance issues related to multi-threaded execution and cache usage are identified and fixes are suggested at a level of detail allowing even novice programmers to perform performance optimization requiring performance experts today.

Among other things, Acumem's technology suggests changes to make cache usage more efficient and to lower memory bandwidth requirements. Most of today's applications use less than half the data brought into the cache. If the applications could be optimized to use memory efficiently, that would lower the cache miss frequency substantially. Other parts of the application would then also benefit from reduced cache pressure. Based on a small application fingerprint file collected from native execution on a system the application's performance on any memory system can be analyzed and application improvements be suggested.

Erik Hagersten
Acumem, Uppsala, Sweden e-mail: erik@acumem.com

Mats Nilsson
Acumem, Uppsala, Sweden e-mail: mats@acumem.com

Magnus Vesterlund
Acumem, Uppsala, Sweden e-mail: mve@acumem.com

1 Introduction

It is a complex and challenging task to take full advantage of today's high performance computer architectures. Their deep memory hierarchies make it difficult to reach the full performance potential, even for the most astute application programmer. Due to the wide and ever increasing memory gap, processors often devote more than half of their time waiting for data to arrive from memory or are stalled due to a congested memory bus. This problem has become worse with the introduction of multicore processors [3]. More concurrent threads contend for the same memory bus bandwidth, and for highly threaded CPUs, the cache size per thread has been seen to decrease. These effects are generally considered to remain the major bottlenecks for many years to come.

Rewriting an application to avoid such bottlenecks requires powerful and insightful performance tools. Until now, performance tools forced developers to wade through a mass of data before they get any idea where the performance problems are. Essentially, the major part of the task – that of identifying and locating bottlenecks – is still the responsibility of the developer. Even then, the process requires as much black art as engineering skill. And perhaps worst of all, it is quite possible to spend a great deal of time trying to identify and fix problems without any guarantee at the outset that there will be any significant performance boost to your application even when the process is completed.

This paper introduces a fresh approach to the problem with a new generation of performance tools based on Acumem's unique fingerprint analysis technology. An application fingerprint is collected at runtime and an off-line analysis can draw conclusions about the applications behavior. This process supports the positive identification and location of application Slowspots and provides suggestions on how to fix them.

This allows the developer to locate and quantify performance problems quickly and simply, displaying a ranking of the potential performance boost for each identified Slowspot. As a result, the productivity of the development process in performance-critical applications is greatly increased, since the developer is presented with an effective cost/benefit analysis of the optimization work. It is of great benefit both to the novice programmer, who is provided with the support to perform advanced optimizations, and also helps the performance expert become more productive by delivering a much higher level of analysis compared to previous tools.

This paper will describe how a new generation of performance analysis tools will automatically identify and locate memory bandwidth-hungry behavior and wasteful cache usage, the two most common problems in performance-critical applications running in multicore environments. It will describe the underlying technology that delivers this breakthrough insight and highlight the key new benefits it will bring to application developers in data-intensive application development.

1.1 It's the Memory, Stupid!

One of the greatest enemies of performance enhancements in the past has been the relatively slow improvement of memory latency [4]. While the processing power of a CPU has become about 4000 times faster compared to 25 years ago, the access time at which data is read from DRAM memory has only improved about four times [1], that is, this so-called memory gap to has grown about 1000 times over this period! This problem is sometimes referred to as the memory wall. Unless the overhead of memory accesses in a data-intensive application can be hidden, the memory accesses will completely dominate the execution time and swamp all other efforts to get performance improvements.

In order to address this memory gap, several levels of caching have been introduced over the years. A cache is a relatively small and fast memory containing a subset of the data present in memory. The cache levels range from a smaller and faster cache (L1), residing close to, and with speeds matching that of the CPU, to larger and somewhat slower caches (L2 and L3) designed to hold larger amounts of data on chip. There is one common objective – performance-limiting accesses to the DRAM must be avoided wherever possible! Following this objective, each new CPU technology generation has historically increased the cache capacity. As a result, the cache capacity devoted to each active thread has generally increased over the last few decades.

In order to effectively take advantage of the fast caches a program must exhibit good data locality. Once a cache line is brought into the cache it should be reused as much as possible before it is evicted again. Since data is brought into the cache in chunks, data that is used at the same time should also be located close together in memory.

Unfortunately, compilers do not address this problem well, as they are instruction centric and typically do not have enough information about data usage available at compile time. For such dynamic data situations the compiler doesn't have enough information, and it is up the programmer to optimize data layout and access patterns to fully leverage the underlying processors' cache.

While locality and its benefit to application performance is widely acknowledged, a typical programmer does not necessarily understand how to write applications to fully exploit these locality properties. Furthermore, there is to date no good way of measuring and reporting this information for an application. This also makes it very difficult for the programmer to measure how well their code has been optimized with respect to these properties.

1.2 Multicore Exacerbates the Memory Problem

Over the past couple of years, CPU frequencies have not improved much; instead, several CPU cores are put on each chip to improve the system performance. These so-called multicore CPUs have the potential to deliver an improvement in execution

capability, but unfortunately the memory interface speed has not kept up. And in data-intensive applications, this is where the performance bottlenecks are likely to be found. The many separate cores have to share the cache capacity of the chip in one way or another. In some implementations, all cores share one common cache at the highest cache level and will have to fight for cache space at execution time. In other implementations, the cache capacity is statically divided between the cores. Furthermore, some CPU implementations allow each core to host several execution threads which reduces the cache capacity per active thread even further [5].

2 Throughput Study of SPEC CPU 2006

It is quite easy to assess how much the system throughput really improves if you utilize all four cores of a quad core system compared with if just one is utilized. Actually, all the needed experimental data are readily available on the web.

The SPEC organization is chartered to collect a representative set of applications to be used in comparison between different hardware and software components. The current set of CPU applications is a collection of 29 widely used applications from different areas.

The applications are divided into the two groups: integer and floating point, depending on what kind of CPU arithmetic's they most commonly use. The SPEC CPU applications can be run in two different ways: Either just a single application is run and the execution time is measured (here referred to as *ordinary* runs), or several instances of each application are run simultaneously and the total time is measured, referred to as *rate* runs.[1]

Among the thousands of official numbers published on the SPEC organization's web page (`www.spec.org`), some of the quad core systems are reported both for ordinary runs and rate runs. It is thus quite easy to compute the throughput improvement for each of the 29 applications enjoy when all four cores are used instead of just one.

Fig. 1 and Fig. 2 show the result of such a comparison of Intel's fastest numbers reported during Q1 2008. The studied system has a Intel Core2 Quad 9550 CPU with 12 MB L2 cache, running at 2833 MHz and connected to a 1333 MHz FSB.

These figures show the relative throughput for each application when running on four cores instead of one. If there was no major competition for the shared caches and memory bandwidth, we should expect the relative throughput to be close to four. As can be seen for the floating point applications, only 8 of the 17 applications experience a relative throughput improvement of more than 2.5. The application with the worst scalability is *470.lbm*.

The integer application seem to perform much better. Here, 9 out of 12 applications experience a throughput improvement larger than 2.5. Still it is a bit disap-

[1] It should stressed that the SPEC rate numbers in no way report the performance of a *parallel* version of the application, since independent copies of each application is run at the same time, not one parallelized copy.

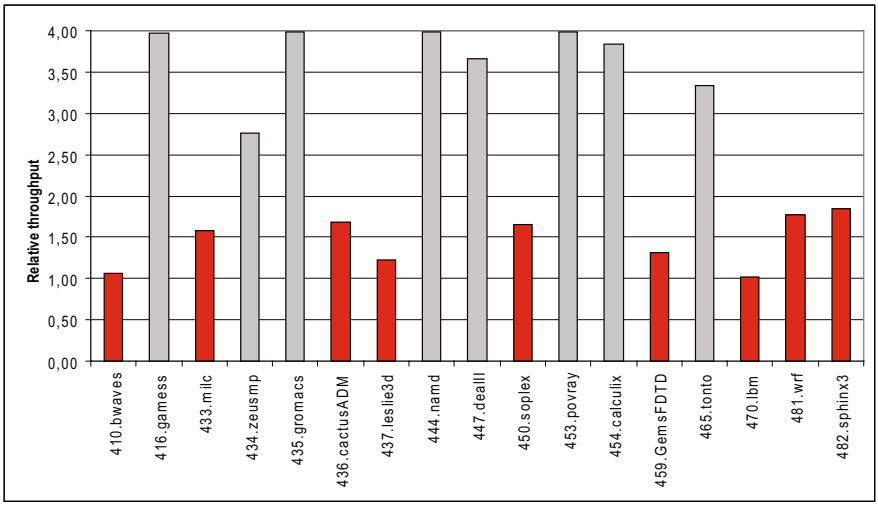


Fig. 1 SPEC 2006 Floating point applications: Relative throughput on four cores compared to one core. 4.0 is linear scalability

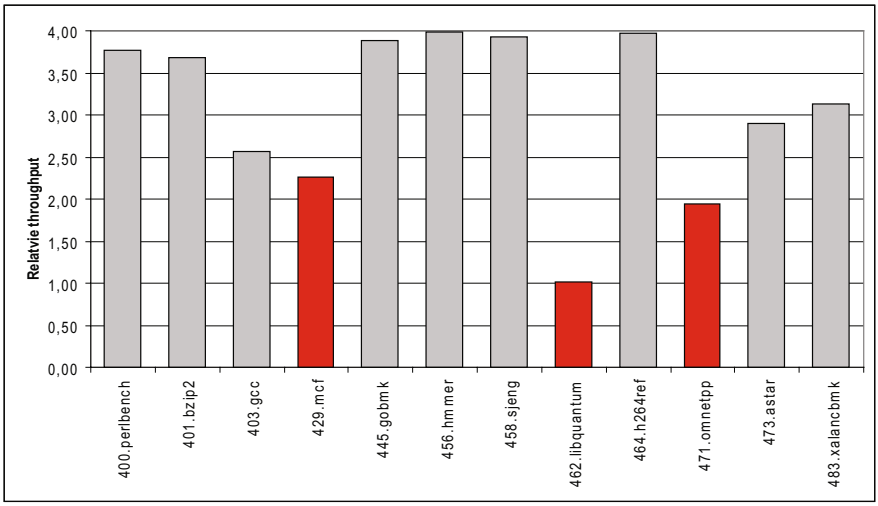


Fig. 2 SPEC 2006 Integer applications: Relative throughput on four cores compared to one core. 4.0 is linear scalability

pointing that almost half of the applications do not reach the 3.0 mark. Here, the worst application is *462.libquantum*.

We can conclude that there are huge throughput and scalability problems for the large set of representative applications collected by the SPEC organization.

3 First Generation Performance Tools Based on Hardware Counters

There are plenty of so-called performance analysis tools available today. Unfortunately, few of them perform any real analysis. Many of them are based on hardware counters, which count specific events taking place inside the processor. These hardware counters can for example count the number of cache accesses and the number of cache misses for each cache level.

Some of the more advanced tools can more or less accurately tell to what degree each source code line contributes to the different event counts.

Since only a limited number of hardware counters can be monitored during a single run, the same execution may have to be accurately repeated several times in order to gather all data. Some tools require the same execution to be repeated up to 37 times to fully collect all the counter information.

Based on hardware counter values, derived properties can be computed. One such interesting derived property is cache miss ratio, which is the ratio of data requests that are not found in the cache. Just like the case of hardware counter values, the derived values can also be mapped to individual source code lines with some precision.

3.1 A Core Dump of Hardware Counter Values – Now Go Figure!

Such a breakdown of, say, miss ratio, tells the programmer about the locations of hotspots, but doesn't tell anything about what the underlying problem is, nor whether the problem can be addressed.

In short, these tools simply gather cache miss information from the hardware counters and dump them into the lap of the programmer. They give you the haystack, and all you have to do is to find the needle.

Interpreting such data requires a high level of expertise in performance analysis on the part of the programmer, as well as a considerable amount of time and effort. First of all, identifying the nature of the cache misses requires a lot of skill. While some cache misses are unnecessary and can easily be avoided by minor modifications to the source code, others can be required by the overall algorithms and are necessary for bringing the data to the cache. The existing hotspot identification techniques can not help you tell one kind of miss from the other.

The source code line that is associated with the hotspot typically contains several memory accesses and it is not obvious which one is to blame, nor is it obvious what modification to the code, if any, would give you a performance improvement. Also, understanding the necessary code changes is often non-trivial. Furthermore, no information is provided on what the potential performance gain would be.

3.2 Incomplete Analysis of Bandwidth Usage

Most CPUs today rely on hardware prefetching to avoid cache misses. The autonomous hardware prefetch logic monitors the memory access pattern of the application, anticipates what data will be requested in the near future and brings them into the cache before they are needed.

In many programs more than half the memory bandwidth is consumed by hardware prefetching. Hardware counter-based performance tools cannot pinpoint the source of this memory traffic, since it is not initiated by a specific instruction.

3.3 Profilers and Simulators

Other tools used for performance analysis are profilers and simulators. Profilers typically use some sampling technology to deduce where in the code an application spends most of its time. The programmer is presented with for example the top ten functions where the execution time is spent, but is given no explanation of why the time is spent there. The programmer will be given limited or no performance information, and will not get any clear guidance on how the code can be improved.

Simulators could potentially give much more insight into the behavior of an application, but these systems typically lack the necessary additional tools to perform this analysis. Furthermore, simulators are far too slow to study the execution of data-intensive applications running full-sized data. Studying an application running reduced data sets will result in incorrect conclusions when it comes to modern memory systems, since the behavior of a cache varies drastically with the application's input data set.

3.4 The Need for a New Generation of Performance Tools

The realization that efficient cache usage is essential for multicore performance and that the low abstraction level offered by current tools cannot provide a productive development environment for data-intensive applications, led to the development of the new performance analysis technology by Acumem. Another driving force was the incomplete bandwidth analysis in existing tools – an absolute must for multicore performance analysis!

A key objective was to create a tool that could automatically perform as much of this analysis as possible, since the vast amount of data and the analysis of data interdependences is difficult to process manually. We need the tool to tell us how efficiently we are currently using our memory system and its caches, and identify the specific Slowspots in the code where improvements are possible. We need guidance on how the Slowspots can be rewritten to improve performance. We need metrics and efficiency data to help us working on the Slowspots in the most optimal order,

as well as measuring the level of improvement from code changes as we go. Last, but not least, we need to the tool to be fast enough to provide a short turn-around time even when running with full-sized input data.

4 Enter: The New Performance Tool

The fingerprint analysis technology developed by Acumem, based on a completely new approach [2], will deliver all of these capabilities and more. First of all, an application fingerprint is gathered from normal execution on a host computer at runtime. The collected information is very sparse, but contains rich dynamic information. The fingerprinting does not rely on hardware counters, since they do not provide enough information. For example, the fingerprint collection may collect information roughly once in every one millionth operation during one execution run. The information allows the analysis to answer questions such as: When will this data be accessed again and by whom? What is the loop structure around these operations? In what context were the operations called? How likely is it that a hardware prefetcher will be able to kick in?

The fingerprint collection is performed on unmodified binaries in a language-agnostic way and works with code produced by literally any compiler. Since the information is collected sparsely, the execution overhead can be kept at a minimum, despite collecting this rich information. Fingerprints from single-threaded as well as multi-threaded execution are collected in a completely transparent way. In fact, the fingerprint collector can attach to an already running process with many threads, collect data for a while and then detach from the process, which will continue its execution undisturbed. The application fingerprint information is captured in a file of typically a couple megabytes size, regardless of the execution time. The longer the execution, the more sparsely it can collect information.

The application fingerprint is not limited to information about what happened on the very piece of hardware where the fingerprint was collected. The collected information is processor and memory architecture independent. It is not affected in any way by the cache organization of the host computer. Instead, it captures the locality properties of the binary which is the product of the source code, the compiler used to compile it and the input data set used to run the application. This enables a fingerprint collected on a specific architecture to be used to analyze the application's behavior on a completely different architecture with a very different memory system – previously only possible using simulation technologies. Contrary to techniques based on simulation, the efficiency of the fingerprint capture allows for full-sized input datasets which is essential for studying real-world problems.

The fingerprint is then analyzed off-line using statistical methods. Target architecture parameters tell the analysis software about the architecture for which to give advice. These involves parameters such as cache sizes, cache line sizes, TLB page sizes and replacement policy in the caches. By varying these parameters, the fingerprint collected on one architecture can be used to suggest application enhancements with respect to a completely different architecture.

This novel approach, based on sparse fingerprints and statistical analysis, enables Acumem VPE to provide much richer and more precise feedback to the user than the more limiting and traditional hardware-counter approach. It can not only identify the exact operations that cause misses in a cache system or those instructions that waste memory bandwidth, it can also tell the reason for the misses and suggest possible fixes, if there are any. Acumem VPE can also often estimate the potential performance gain, which is very helpful in determining if a fix to an issue should be attempted.

A Slowspot is a piece of code with an identified performance problem that can be improved in a way that would result in a substantial performance gain.

For each Slowspot, the tool reports the nature of the inefficiency and the set of suggested fixes to the application. The analysis can also estimate the overall improvement if all Slowspots were fixed, as well as estimate the potential improvement for each individual Slowspot.

The analysis technology can do a lot more than just assist in performance optimization. For example, when planning a possible move to new processor architectures, it can also provide a performance comparison for the existing code for a range of candidate processor architectures based purely on analysis performed by running the existing application code on the current system. It can even identify the Slowspots and how to address them for this future processor architecture.

4.1 Slowspot Insights

For each identified Slowspot, the improvement potential is assessed, answering complex questions such as; What fraction of the total cache misses would be avoided if I fixed this Slowspot? How much would my memory bandwidth requirement decrease? These numbers can be a quick guide when choosing the Slowspot to attack first. Other performance insights provided by the technology will further enable a more systematic way of improving application performance. A typical approach could include the following steps:

1. The Slowspots caused by ineffective cache line utilization can be assessed and corrected.
2. Slowspots with data reuse opportunities can be identified and fixed to avoid cache misses by maximizing reuse of data in the cache.
3. Slowspots, where the hardware prefetcher is unlikely to be effective can be assessed, and alternative algorithms or data placement methods can be selected.
4. Useless instructions can be identified and removed.
5. The technology can guide the insertion of software prefetch instructions for remaining places causing cache misses not avoidable by other means

4.2 Improving Cache Line Utilization

Even though the caches play such an important role for performance, most applications devote more than half of the cache capacity to store unused data, that is, more than half of the data brought into the cache is not ever used before it is replaced. This is a fact most programmers seem blissfully ignorant of.

Acumem VPE introduces the term *cache line utilization* to indicate the fraction of the data brought into the cache which is actually used. Poor cache line utilization implies that the application is wasting both memory bandwidth and cache space. An examples of this is a cache line of 64 bytes brought into the cache of which only one word of 8 bytes is needed, that is, a cache line utilization of 12.5%. Poor cache utilization is a prime example of Slowspot behavior.

Following the tool's guidance and rewriting the program in a way that improves the cache utilization would have three positive effects:

1. Cache misses would occur less often, since each cache miss would now bring in more pieces of useful data.
2. The amount of data an application moves across the memory bus would decrease.
3. The system would behave as if there were more cache available after the rewrite, since the same amount of useful data can now fit in a smaller part of the cache. This will allow for even more popular data to reside in cache, which can further reduce the cache miss ratio and the amount of data moved to/from the memory.

The analysis can not only find poor cache line utilization for data brought into the cache by cache misses, but also for data retrieved by hardware prefetching as well as by software prefetch instructions. The analysis will associate these Slowspots with the source code lines consuming the retrieved data and suggest how the can be rewritten.

There can be many different reasons for poor cache line utilization, some of which are described below:

Table 1 Examples of reasons for poor cache line utilization

Reason	Description
Loop interchange	A multidimensional data structure is traversed in a non-optimal dimension order (a.k.a. incorrect loop nesting)
Sparse data allocation	The data allocation creates a data structure with unused holes.
Sparse data usage	A loop only uses every *n*th data element of a data structure, for example, only accessing one member of a combined data structure, such as a "struct".
Random access pattern	Pointer-based or tree-based accesses may access the data space in a random manner.

The analysis can often identify the reason for the poor cache line utilization for each Slowspot. It can also identify the data structure that is involved with each specific Slowspot. The analysis can report the average cache line utilization for the

entire application, for each loop and for each Slowspot. The analysis could also calculate the potential memory traffic reduction if the cache utilization was improved.

It is interesting to note that applications with poor cache behavior often have poor cache utilization. This is fortunate, since fixing poor cache utilization is often a fairly simple task, once the location and nature of the problem is known. About 70% of the SPEC CPU2006 benchmarks with a high cache miss ratio have a cache utilization of 50% or lower. Some of them have cache utilization as low as 10-20%! Improving cache utilization from 50% to 100% would generally result in an execution with less than half as many cache misses and make less than half as many accesses across the memory bus bottleneck. Our experience shows that this can more than double throughput in a multicore system.

4.3 Data reuse Opportunities

Another example of Slowspots identifiable by analysis concerns data reuse. Often, applications repeatedly access the same data structure over and over again, but the accesses are so far apart that the data has been replaced in the cache prior to its reuse. It may be that different rewriting techniques can be used to move the data reuse closer in time to allow for the data to remain in the cache at the point of reuse. Identifying such opportunities requires a global view of when the data structure is accessed, the cache behavior and the existence of any data-dependent accesses that would prevent such a rewrite. Acumem's technology can perform all these three steps automatically and identify Slowspots with probable data reuse opportunities.

Table 2 Examples of data-reuse Slowspots

Slowspot	Description
Tiling (a.k.a. blocking)	A loop is reusing data, but by the time the reuse occurs data has already been evicted from the cache. By processing a smaller piece of the data set at a time the data can be reused while it is still in the cache.
Temporal loop fusion	Two different loops are using the same data. By merging the loops the data only read into the cache once.
Spatial loop fusion	Two different loops are using data from the same cache line. By merging the loops the data only read into the cache once.

4.4 Guiding Prefetch Instruction Insertion

Most architectures provide software prefetch instruction, allowing for a compiler or a programmer to explicitly bring a piece of data into the cache prior to its use. Correctly inserted software prefetch instructions can help remove cache misses for which there is no other apparent fix.

Slowspots related to prefetch instructions that can be diagnosed by the analysis include:

- A prefetch instruction executed too far ahead of actual data use will not help performance since the data will get replaced before the use. Instead it will hurt performance by bringing in the cache line an extra time and thereby wasting memory bandwidth.
- A prefetch instruction executed too close to the actual data use will not provide the full benefit, since the processor does not have time to finish the fetch before the data is needed. The distance between the prefetch and the data use should be increased to give the process more time to finish the fetch.
- A prefetch instruction that repeatedly tries to bring in data that is already present in the cache is harmful to performance since it will use up execution resources without doing any useful work.

5 Utilization Study of the Worst SPEC CPU 2006 Applications

The importance of good cache line utilization can be demonstrated by looking at some curves collected from the worst SPEC CPU FP and the worst SPEC CPU INT applications.

Fig. 3 shows how the fetch ratio (solid red) for the libquantum application changes as a function of cache size. For a system with a total of 12 MB L2 cache running four instances of this application, such as the system used for our throughput

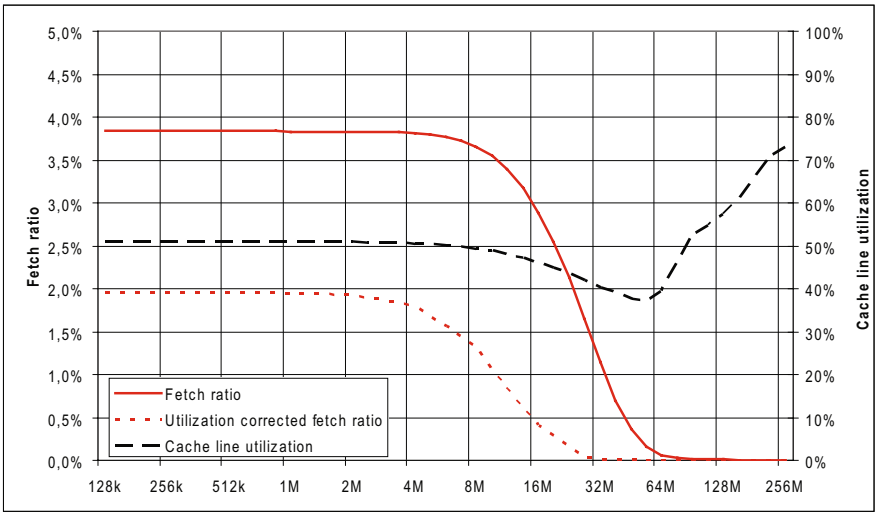


Fig. 3 Cache characteristics of the integer application *462.libquantum*. Fetch ratio scale to the left, cache line utilization scale to the right

analysis, each instance of the libquantum would get 3 MB of cache and according to the diagram have a fetch ratio close to 4%. This means that 4% of all memory accesses would cause data to be fetched from memory, either directly or through prefetching.[2]

This is enough to saturate the memory bus, resulting in poor scalability when running multiple instances on a multicore processor. The poor cache line utilization of the application, shown by the dashed black curve, is a strongly contributing factor. It shows that only half of the available cache capacity is used for 3 MB cache (the scale is to the right of the chart). The red dotted red curve shows the estimated fetch ratio if the application is rewritten in a way that improves the cache line utilization to 100%. At 3 MB, the pressure on the memory bus would be cut by more than a factor two down to a miss ratio of 1.9%.

Fig. 4 shows a similar graph for the worst floating point application from the throughput study: *470.lbm*. Once more, we see a fetch ratio in the 4% range and cache line utilization close to 50% for a 3MB cache. Fixing the cache line utilization would almost save a factor two on the memory bandwidth, reducing the fetch ratio from 4.2% down to 2.1%.

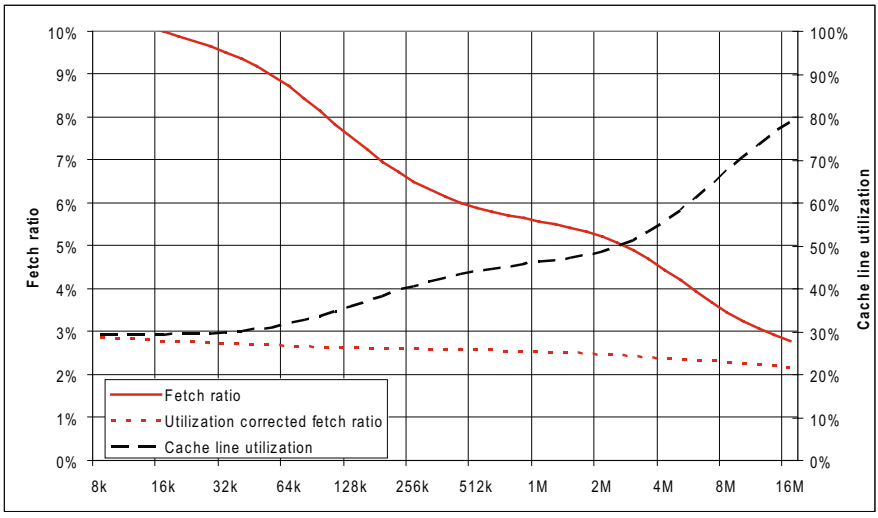


Fig. 4 Cache characteristics for the floating point application *470.lbm*. Fetch ratio scale to the left, cache line utilization scale to the right

[2] The term *miss ratio* is reserved for the fraction of accesses that would cause memory to be fetched, but for which a prefetcher would be ineffective, meaning that those accesses would also cause a CPU stall in addition to consuming memory bus bandwidth.

6 Tuning Example: 179.art

179.art is an application included in the SPEC CPU 2000 benchmark suite. It is an excellent demonstration application for performance analysis since it is guilty of almost all the performance mistakes in the book.

6.1 Obtaining the Fingerprint

The first step is to obtain the fingerprint. Assuming a default installation and a PATH environment variable properly setup, type:

```
$ sample -o art.smp -r ./art [parameters to art]
```

This starts the application and attaches the fingerprint collector from the very start. In this mode, it will continue to collect fingerprint data until the program terminates normally, after which some postprocessing of the fingerprint will take place.

There are many different command line options to control when and how to start and stop collecting data to cover the part of the program you are interested in. It is often sufficient to let the collector run during just a representative subset of the applications execution. As collecting a fingerprint adds to the execution time, it is useful to quickly be able to get a coarse report from a limited fingerprinting.

6.2 Preparing a Report

The fingerprint is now in the file `art.smp`. It contains representative information for the application and data set, independent of the current memory hierarchy. When preparing the report, we also add information about the cache for which we want to tune performance.

```
$ report -i art.smp -o art-report -c 512k
```

The `report` command prepares a report for a target cache of 512 kbytes. The report will list those issues that are relevant for this particular cache size. Varying cache parameters will give different reports, as different problems are exposed on different architectures.

Bring up the report using your web browser:

```
$ firefox art-report.html
```

6.3 Interpreting the Report

Fig. 5 shows one page from the report. It is divided into three sections. The top left section is for summaries and tables of issues and loops. The bottom left section is for loop and issue details. The larger section to the right is for displaying annotated source code.

The top latency Slowspot for this case indicates that a data structure suffers from *inefficient loop nesting*, i.e., it is traversed along the wrong dimension in the innermost loop. Acumem VPE helps identifying the variable `bus` in the following code snippet:

```
for (tj = 0; tj < numf2s; tj++)
{
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

The problem is that the loops iterate over the `bus` matrix in the wrong direction, along the columns instead of along the lines. The means that only one element from each cache line is used in each iteration in the inner loop. The matrix contains more

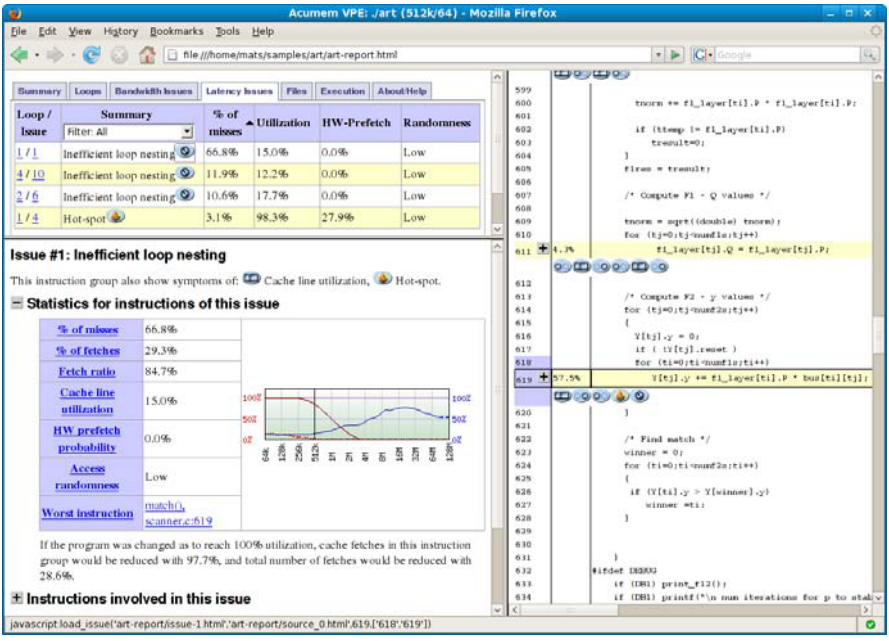


Fig. 5 Screen shot of Acumem VPE showing the worst latency Slowspot (Inefficient loop nesting) and the corresponding source code

lines than fit in the cache, so by the time the iteration starts over from the top of the matrix those cache lines have already been evicted, causing every memory access to miss in the cache. Changing the way `bus` is accessed by transposing it fixes this problem. The actual `for` statements will remain intact in this case, but in other cases it makes more sense to reorder the loop statements. Acumem VPE also gives a prediction on how many fewer memory fetches this application will perform after fixing this problem (30%).

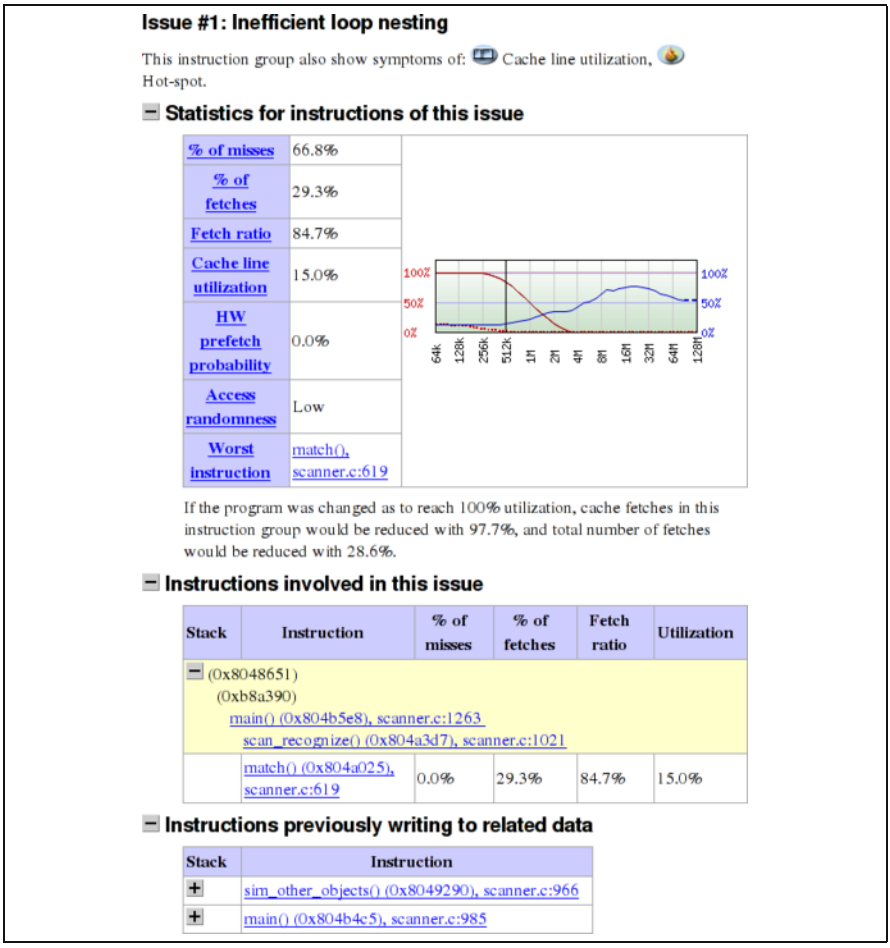


Fig. 6 Acumem VPE report for the Slowspot *inefficient loop nesting*

Incidentally, there is something else wrong with this line. The second bandwidth related Slowspot tells us that the *cache line utilization* of `f1_layer[ti].P` is poor, and not every byte is used in the cache lines where this data is stored. In fact, it tells us that about 8 out of 64 bytes (13.7%) in each cache line is ever used in this loop before that cache line is replaced in the cache with some new data.

The problem is that only one field (`.P`) is ever used in this loop, but there are several other fields of this structure occupying adjacent memory locations. Acumem VPE estimates that 26% of the application's memory traffic can be avoided by packing this data differently. The fix is to replace the use of an array of structs with separate arrays for each of the struct's fields.

This particular code section was changed to read:

```
for (tj = 0; tj < numf2s; tj++)
{
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer_P[ti] * bus[tj][ti];
}
```

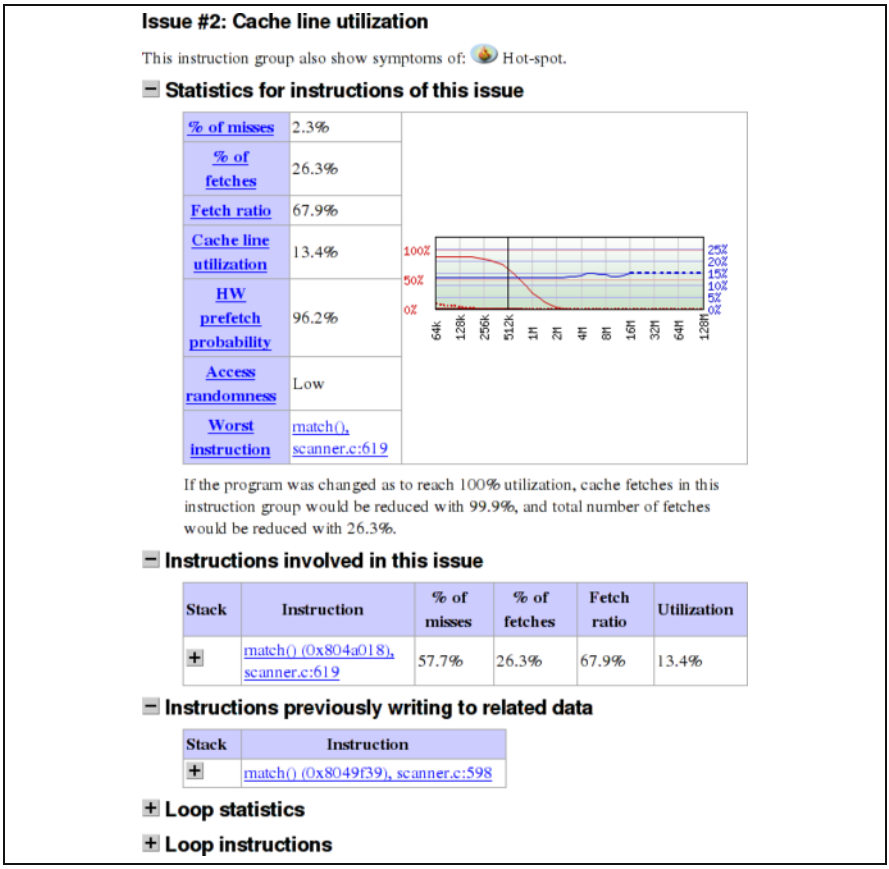


Fig. 7 Acumem VPE report for the *cache line utilization* Slowspot

After fixing these and similar problems througout the code, this program ran 6 times faster on an AMD Athlon 64 X2 3800+ processor.

7 Tuning Example: Revisiting the Throughput Applications

Much like the tuning of *179.art*, both of the misbehaving throughput applications *462.libquantum* and *470.lbm* experience huge improvements based on the suggestions pointed out by Acumem VPE.

For instance, the top Slowspot in *462.libquantum* is presented by Acumem VPE like this:

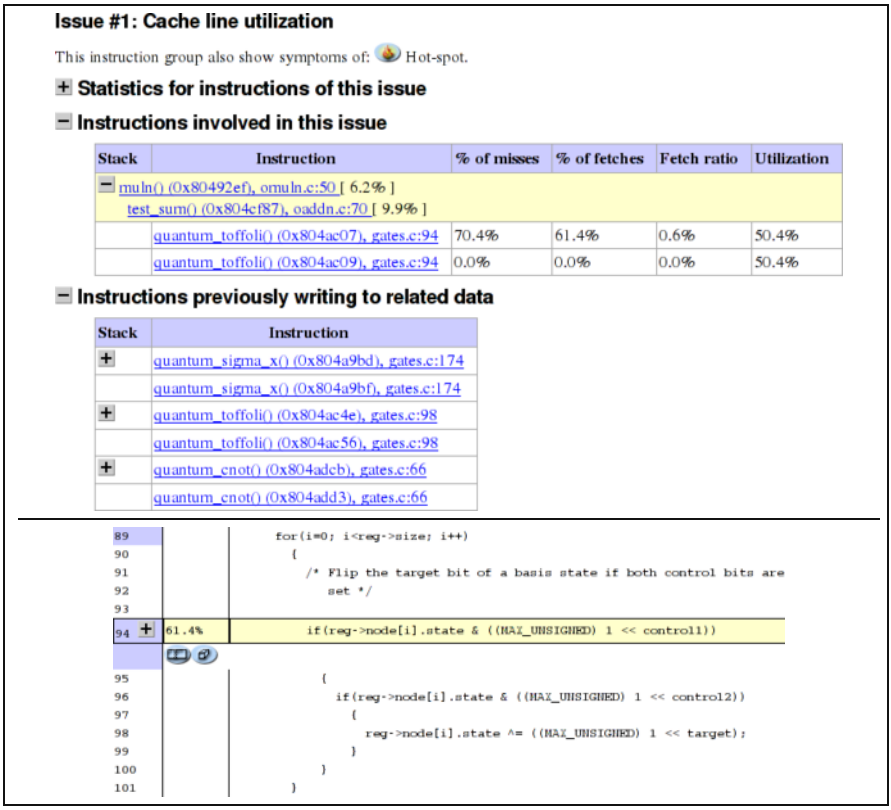


Fig. 8 Top Slowspot in *462.libquantum*

Again this is a case of accessing just one field in a struct. Fixing this problem throughout the application (changing approximately 40 lines of code, separating the fields of the struct into different arrays) causes the scalability to be greatly improved.

In *470.lbm* the way a three dimensional structure was allocated led to a hard to spot invalid loop order. Just eleven lines were changed in a set of macros to fix this issue.

In Fig. 9 and Fig. 10 the result of these optimization can be seen. For both these applications, cache line utilization were improved to close to 100%, and the resulting fetch ratio approached the fetch ratio predicted in Fig. 3 and Fig. 4.

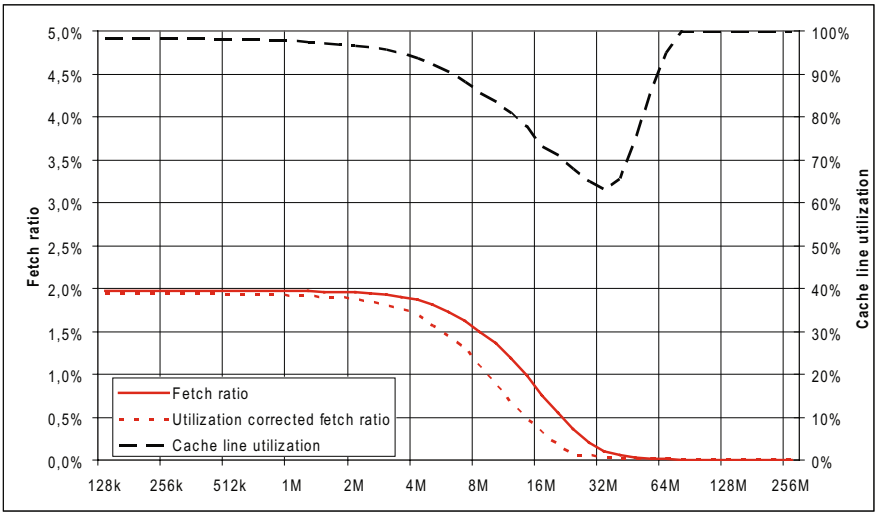


Fig. 9 Characteristics of corrected integer application *462.libquantum*

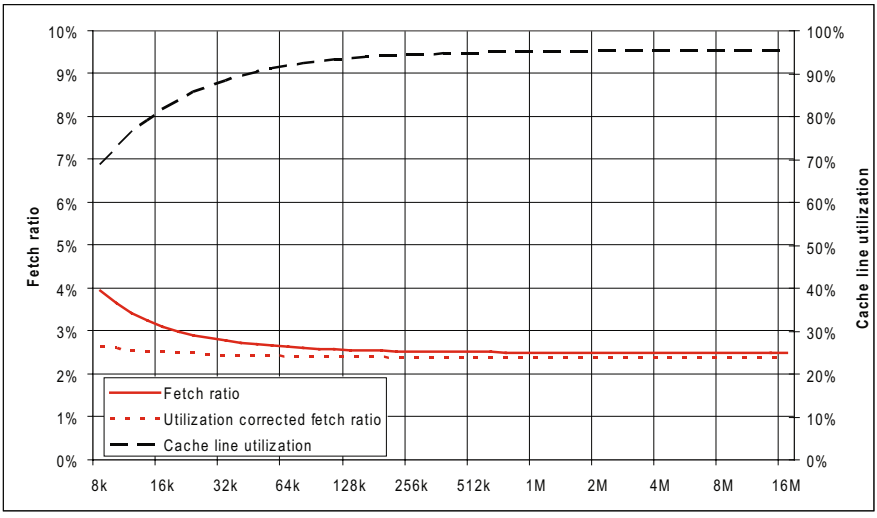


Fig. 10 Characteristics of corrected floating point application *470.lbm*

We compared the execution time of the original and modified versions of the two applications on a Intel Core2 Quad Q6600 2.4GHz, 1066MHz FSB, equipped with 800 MHz DDR2 RAM.

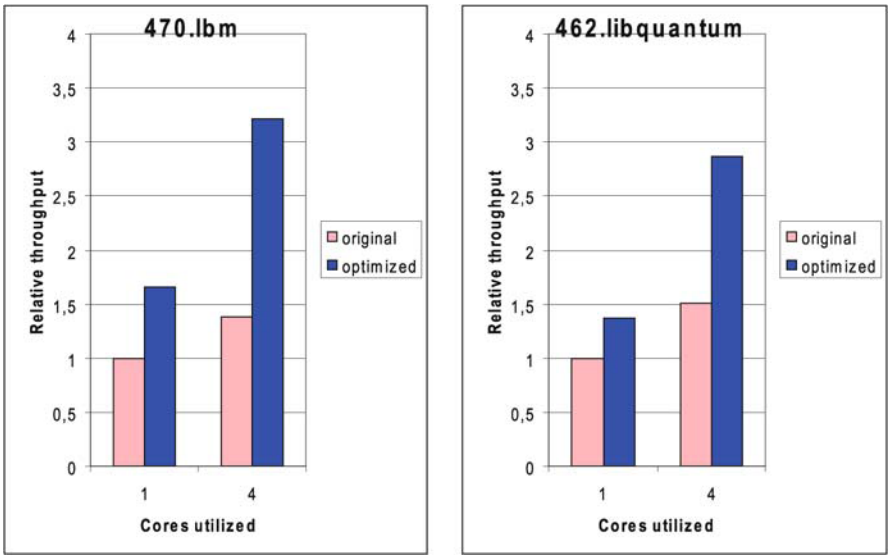


Fig. 11 Scalability comparison of original and optimized versions of applications *470.lbm* and *462.libquantum*

8 Conclusion

Historically, performance analysis tools have performed surprisingly little analysis, and have not done a great deal for performance either, at least not without major expenditure of time and effort on the part of the developer of data-intensive applications. This chronic problem is becoming even worse with the introduction of multicore processors.

Acumem's VPE fingerprinting technology provides a fundamentally different approach to the problem. Firstly, an unprecedented level of detailed information is provided on application performance across a range of processor architectures. Secondly, the detailed analysis is done by the VPE tool, rather than by the developer, providing the location and nature of specific performance Slowspot in the code.

References

1. Hennessy, J.L., Patterson, D.A.: Computer Architecture – A Quantitative Approach. Morgan Kaufmann Publishers, San Francisco, USA (2007)
2. Berg, E., et al.: Fast Data-Locality Profiling of Native Execution. Proceedings of the International Conference on Measurement and Modeling of Computer Systems, Banff, Alberta, Canada (2005)
3. Hammond, L., et al.: A Single-Chip Multiprocessor. IEEE Computer 30(9): 79-85 (1997)
4. Fernandes, E.S.T., et al.: Instruction usage and the memory gap problem. In Proceedings of 14th Symposium on Computer Architecture and High Performance Computing 2002
5. Karlsson, M., et al.: Conserving Memory Bandwidth in Chip Multiprocessors with Runahead Execution. In Proceedings of IPDPS 2007

Parallel Performance Analysis Tools

The Vampir Performance Analysis Tool-Set

Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel

Abstract This paper presents the Vampir tool-set for performance analysis of parallel applications. It consists of the run-time measurement system VampirTrace and the visualization tools Vampir and VampirServer. It describes the major features and outlines the underlying implementation that is necessary to provide low overhead and good scalability. Furthermore, it gives a short overview about the development history and future work as well as related work.

1 Introduction

Performance analysis is an important part in the development and optimization of parallel High Performance Computing (HPC) applications. In order to achieve close to optimal performance one needs to understand the dynamic run-time behavior in detail. In particular, on today's more and more complex hardware architectures including multi- and many-core CPUs and growing hierarchies of caches as well as local and remote main memory.

Therefore, it is relying on sophisticated tools in order to give the user insight about fast, complex, and highly parallel dynamic run-time behavior.

This paper gives an overview about the Vampir tool family which is widely known in the national and international HPC community. It consists of the instrumentation and measurement component VampirTrace and the visualization applications Vampir and VampirServer. The most important established and new features of both components will be presented.

VampirTrace was originally forked from the KOJAK trace library [17] and is available as open source software under a BSD license. The visual analysis tool Vampir has a history of over 12 years. Starting at FZ Jülich it is now being developed at ZIH, Technische Universität Dresden. VampirServer, which is the parallel

Andreas Knüpfer
ZIH, TU Dresden, 01062 Dresden, e-mail: andreas.knuepfer@tu-dresden.de

successor of Vampir, was first released in 2003. Vampir has always been a commercial product, formerly distributed by Pallas GmbH, today both versions, Vampir and VampirServer are commercially available via the GWT mbH Dresden.

The rest of the paper is organized as follows: After an introduction to profile-based and trace-based performance analysis the three main parts cover instrumentation of user applications, run-time measurement and visual analysis. Finally there is an overview on related work. Then conclusions are given together with an outlook to current and future development.

2 Performance Analysis via Profiling or Tracing

Performance analysis and optimization of applications are important phases of the development cycle. Like testing and debugging it should be imperative, at least for High Performance Computing (HPC) applications. It is an important precondition to guarantee efficient usage of expensive and limited computing resources in general. This means obtaining the results with minimum resource usage and costs.

Furthermore, it is important for scalability, i.e. being able to achieve the next bigger simulation with the given resources. For that, it is particularly important to exploit as much of the theoretically available performance as possible.

The task of the performance analysis phase is firstly, to measure the actual performance on a given platform in terms of computing speed or throughput as well as resource consumption in terms of run-time or memory requirement or storage space. Secondly, performance analysis has to identify opportunities for performance improvement or reduction of resource usage.

Like debugging, i.e. error analysis, the process of performance analysis is inherently difficult and human users profit much from tool support. Ignoring tools altogether in favor of manual *printf*-like performance analysis is strongly inadvisable from the point of view of software engineering, overhead, scalability and usability.

There are two well known approaches for sequential and parallel performance analysis tools: *profiling* and *tracing*. In general, profiling collects aggregated information about certain *events* during a program run, whereas tracing records information about individual *events*. The events are simply points of interest in the course of program execution. The most common event types are:

- enter and leave, i.e. call and return of functions or subroutines
- send and receive, i.e. point-to-point message passing operations
- collective communication operations as known from the MPI standard
- performance counter samples, that provide a scalar value at a point in time

Information about events allow to infer about application flow like function calls (or general basic blocks), communication or other activities relative to individual processes or threads. With profiling it is possible to collect summarized information about function calls etc. over the total program run-time or separated for certain

phases of it [4, 6, 14, 20]. Typically, this includes total run-time per function, number of calls and the call tree, i.e. caller-callee relations between functions.

On the other hand, tracing records all individual events along with general properties like exact time stamp and the particular process or thread as well as further event type specific properties. Thereby, tracing allows to investigate single events. This enables trace-based tools to identify variation in the dynamic behavior of a single function over many iterations, which profiling could not detect. Furthermore, profiles can be computed from traces but not vice versa. This comes to the price of larger result data sets and increased overhead.

Both approaches, profiling and tracing, require modification of the target application in order to detect event occurrences. This step, which is also called instrumentation, follows the same principles for both methods.

3 Instrumentation with VampirTrace

There are several ways for instrumentation of target applications which are more or less suited for different programming constructs and different event types. VampirTrace allows automatic instrumentation for the following programming paradigms:

- sequential programs,
- MPI programs,
- OpenMP programs and
- hybrid MPI and OpenMP programs.

The most prominent ways of instrumentation used by VampirTrace are:

- compiler instrumentation,
- source-to-source instrumentation,
- library instrumentation and
- manual instrumentation

which are performed at build time. The VampirTrace instrumentation software provides convenient *compiler wrappers* that take care of most instrumentation details. Similar to the well known MPI compiler wrappers, they refer to underlying platform compilers for code generation. At the same time, command line parameters are added (e.g. for instrumentation), additional libraries are linked (e.g. the measurement library) or further commands are executed during the normal build process (e.g. pre-processing the source files for automatic source-to-source instrumentation).

From the user perspective this requires merely replacing compiler commands, for example in corresponding Makefiles like the following:

```
CC= gcc              CC= vtcc
CXX= g++             CXX= vtcxx
F90= gfortran        F90= vtf90
```

The compiler wrappers will select the instrumentation method suitable for the particular platform and the specific instrumentation options for the underlying compilers [9, 18]. See the following sections for more elaborate descriptions of the instrumentation methods of VampirTrace.

3.1 Compiler Instrumentation

The automatic instrumentation using the compiler is the most convenient way to instrument an application. Special flags cause the compiler to generate instrumentation calls for entries and exits to/from functions. The measurement functions will be called just after function entry and just before function exit. Currently, VampirTrace supports following compilers for automatic instrumentation[1]:

- GNU compiler collection (gcc, g++, gfortran)
- Intel compiler version 10 (icc, icpc, ifort)
- PGI compiler (pgcc, pgCC, pgf77, pgf90)
- IBM compiler (xlc, xlC, xlf77, xlf90)
- Sun Studio compiler (Fortran only)
- NEC SX compiler

This automatic approach for instrumentation of user functions is very convenient but also comes with one disadvantage. Usually, all functions will be instrumented and traced, thus the resulting tracefile can easily become very large. Additionally, the runtime of a fully instrumented application can significantly increase, because the measurement overhead for instrumented functions is large compared to the execution time of very small uninstrumented functions. To avoid these negative effects, it is recommended to exclude (filter) frequent short function calls, see also Sect. 4.6.

Compilers have different behaviors when automatically instrumenting inlined functions. For example, the GNU and Intel compilers switch off inlining completely when requested to insert hooks. Other compilers still perform inlining (depending on the optimization level), but do not insert hooks in those functions. The bottom line is that you cannot inline and instrument a function at the same time.

3.2 Source-to-Source Instrumentation

Automatic source-to-source instrumentation is an alternative approach to compiler instrumentation. VampirTrace uses this method for instrumentation of OpenMP directives via the OPARI [13] pre-processor for C, C++ and Fortran. It automatically inserts instrumentation calls to the POMP profiling interface in a copy of the original

[1] All but the GNU and the Intel compiler instrumentation of user function entries and exits is based on undocumented and unsupported compiler options.

source files. VampirTrace provides appropriate POMP profiling functions to detect OpenMP events including context information and source code locations.

3.3 Library Instrumentation

Library instrumentation replaces an existing library with an instrumented counterpart. This requires substantial effort because for all API functions a replacement needs to be provided. Therefore, it is only recommended for standard libraries. VampirTrace uses this technique to monitor MPI (Message Passing Interface) calls.

This method has the advantage that function arguments are visible to the measurement system and can be evaluated, for example to determine communication peers within MPI calls. Furthermore, it allows to insert instrumentation of the particular library without re-compilation but with re-linking or with dynamic linking.

For the functionality of instrumented libraries it is most desirable to refer to the original library. Because the original function symbols are shadowed by the instrumentation symbols, they have to be resolved explicitly via the dynamic loader library. This is not necessary for MPI however, as the MPI standard guarantees alternative symbols for all API functions, compare Fig. 1.

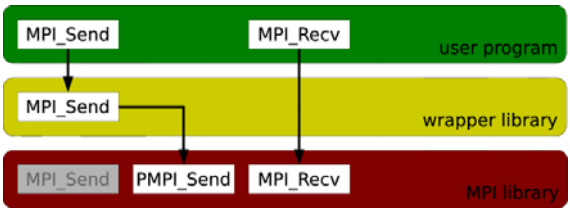


Fig. 1 Library instrumentation for MPI. The wrapper library intercepts all API calls in order to detect run-time events and refers to alternative symbols in the underlying original MPI library

3.4 Manual Source Instrumentation

In addition to the automatic methods manual instrumentation is supported. The VampirTrace API provides several calls which can be used to mark functions or any user-defined source code regions.

This allows more detailed control which functions to include in the instrumentation and which to exclude in order to reduce overhead. Furthermore, the manual instrumentation can be combined with all other ones. For example, all user functions can be instrumented by a compiler while extra source code regions like loop bodies can be instrumented manually by using the VampirTrace API.

4 Run-Time Measurement and Event Recording

The run-time measurement part of the event tracing software includes several components that need to work together in order to provide a consistent view on dynamic application behavior. In particular, all components need to pay close attention to minimize measurement overhead.

4.1 Timer Synchronization

The performance analysis of a parallel program requires a global view on the timing information of events. This is provided implicitly on systems equipped with a fine granular global clock. Distributed system architectures, where every processor is equipped with a local clock only, like cluster environments, require explicit clock synchronization. Even though, the clocks of such systems might be synchronized via network protocols like NTP which is not accurate enough for tracing with high resolution timers like CPU cycle counters.

VampirTrace currently exchanges clock synchronization information at the very beginning and the very end of a trace run. Still, during tracing every process records all local events according to the local asynchronous timer. In a post-processing step, the time-stamps are translated to a global timer, i.e. one local timer selected as the master. Differences in offset and speed between the local timers are interpolated linearly between the initial and final synchronization point.

4.2 Recording of Hardware Performance Counters

Many processors provide hardware performance counters that expose fine-grained information about the processor's inner workings. There exist counters which show the number of executed floating point operations, the number of cache misses for the various cache levels, statistics on branch instructions, etc.

The available hardware performance counters can be recorded by VampirTrace during the tracing run. This is accomplished by using the PAPI library for accessing the counters on supported processors. Selection of the counters to be saved is done by merely setting an environment variable. This allows a user to easily carry out multiple tracing runs that capture different hardware performance counters.

4.3 Recording Application's Memory Usage

VampirTrace is able to track the dynamic memory usage of an application [10]. It uses GNU glibc's special hook mechanism to exchange the original memory al-

location functions `malloc`, `realloc` and `free` with wrapper functions[2]. The wrappers save the amount of allocated memory as counter records which allows the user to easily examine the information from within Vampir.

As a drawback from the current implementation it is not possible to trace memory usage of multi-threaded applications because the hook mechanism is not thread-safe. Although an implementation could introduce thread-safe behavior by using explicit locking, this has not been done to avoid the extensive overhead which lies therein.

4.4 I/O Activity Tracing

Like memory allocation functions, POSIX I/O functions can be intercepted by VampirTrace, too [15], see also Table 1. If this feature is activated, then each invocation of an I/O routine will be recorded in the trace along with the respective I/O event. For intercepting the calls VampirTrace overwrites all I/O API functions with own wrapper functions.

At run-time VampirTrace determines the addresses of the original I/O functions in the Standard C Library via `dlsym()` and calls them to execute the actual I/O operations. Afterwards an I/O event record is created which holds timing information as well as I/O-specific data.

The details of individual I/O calls can then be investigated with Vampir, see also Sect. 5.2. Furthermore, I/O performance counters can be generated from the events by a post-processing tool. This allows a quick overview of I/O activities over the run-time, see Fig. 5.

Additionally, performance data from external sources, e.g. SAN controllers, can be integrated for evaluating application I/O performance with regard to the total system I/O activities. However, this is not portable as it requires non-standard APIs to query aggregated I/O statistics as well as special access privileges [15].

Table 1 List of POSIX I/O functions captured by VampirTrace

`open`	`open64`	`creat`	`creat64`	`close`	`dup/dup2`
`lseek`	`lseek64`	`fdopen`	`fopen`	`fopen64`	`fclose`
`fseek`	`fseeko`	`fseeko64`	`fsetpos`	`fsetpos64`	`rewind`
`read`	`write`	`readv`	`writev`	`pread`	`pwrite`
`pread64`	`pwrite64`	`fread`	`fwrite`	`fgetc`	`getc`
`fputc`	`putc`	`fgets`	`fputs`	`fscanf`	`fprintf`

[2] This mechanism is not limited to GNU glibc but it indeed is available on other libc implementations, e.g. SGI IRIX.

4.5 User Defined Performance Counters

The VampirTrace API provides another type of manual instrumentation calls which allow the recording of user defined values, e.g. loop iteration counts, calculation results, residuals of iterative solvers, or any other scalar quantity. This can be used to visualize performance data and application data in one go. A user counter is defined by its name, a counter group it belongs to, the value type (integer or floating point), and the unit. During run-time the application can record counter samples at any convenient time.

4.6 Grouping and Filtering of Functions

Grouping of function symbols allows to determine sets of associated functions. This might be grouping by functionality or by affiliation to certain classes or packages or libraries. The group specification has to be defined before run-time in a plain text file. Function names are assigned to groups, wildcard clauses are allowed. VampirTrace knows default groups for MPI calls, OpenMP functions, I/O functions, and memory allocation calls. During the visualization phase the groups are represented by different colors. Furthermore, aggregated statistics can be retrieved for all functions of a group.

As mentioned earlier, tracing with automatic instrumentation may result in substantial overhead as well as huge amounts of trace data. The filtering feature of VampirTrace provides an ability to control this with a simple exclude list of functions not to be recorded. Furthermore, function specific limits can be assigned to record not more than a certain number of occurrences. Again, this is specified in a plain text file at run-time.

4.7 Event Buffering, Flushing & Trace File Format

During run-time, every process or thread is storing event records to a memory buffer of adjustable size. Only by this means, it is possible to achieve low overhead per event. Yet, the buffer is taken from the memory size available to the application. Therefore, its size needs to be selected carefully. As soon as this buffer is exceeded it is flushed to a trace file. Then event recording is either discontinued or resumed.

The former is the default behavior, since it avoids producing unforeseeable large traces by mistake. Via run-time parameters the size of the buffers can be adjusted, as well as the maximum number of flushes allowed per process. In the latter case, the intermediate flush phases are marked in the event trace itself, as it may cause substantial interference with the execution of the application. Since it cannot be avoided altogether, it has to be recognizable in the analysis, at least.

One way or another, the trace data of every process/thread is written to a separate file in the Open Trace Format (OTF) eventually [11]. This file may be located on a local file system within a distributed environment. In the course of an automatic post-processing all local trace files are moved to a common global location and joined to a single OTF trace consisting of multiple files (streams).

4.8 Run-Time Overhead

Tracing inevitably causes overhead that slows down the execution of the application and alters the original behavior of the application to be monitored. Therefore, the tracing infrastructure should try to minimize this effect such that:

- the recorded timing resembles the original application as close as possible and
- the overall run-time of the traced application stays inside an acceptable range.

Run-time overhead is introduced during four parts of the tracing:

- initialization at program start-up
- per-event overhead (in event handlers)
- storage of trace data to disk (buffer flush)
- finalization

The initialization sets up internal data structures, gets symbol information, performs early timer synchronization, etc. This overhead accounts before the actual start of an application and therefore usually is noncritical.

The time to call individual event handlers contributes the most part of the critical overhead of tracing. The per-event overhead does not depend on the duration of the recorded event, thus significant overhead is produced for very frequent short events.

Storing trace data to files produces considerable overhead as well, see also Sect. 4.7. By default, VampirTrace will postpone this operation to the finalization phase. Then it is outside the scope of the original program run-time and thus noncritical for performance analysis. If this is not feasible, intermediate flush operations are necessary. This causes substantial perturbation to the course of execution, which is marked for consideration during analysis. This is even worse for parallel execution, because intermediate flushing is triggered by the individual processes/threads in an uncoordinated manner.

During the finalization the (remaining) event buffer contents are flushed to files. In addition some post-processing of the trace data is required, including unification of local identifiers and tokens and time correction. This costs additional effort (in computation and I/O) but does not influence the quality of the measurement anymore.

For a real-world estimation the run-time overhead per event has been evaluated for function call events, see Table 2. It shows the overhead per function call for an instrumented application with different measurement settings. All times have been

Table 2 Function call overhead introduced by VampirTrace (excl. original function run-time) on a SGI Altix 4700 system with 1.6 GHz Intel Itanium II CPUs

tracing mode	overhead per call
filtered out	0.82μs
recorded	0.92μs
with one PAPI counter	4.47μs
with 3 PAPI counters	4.61μs

measured on an SGI Altix 4700 which employs Intel Itanium 2 processors with 1.6 GHz speed. Function inlining has been disabled explicitly.

Minimal overhead occurs when the executed function is filtered out. Recording the function call including timer etc. costs minimal additional overhead only. Accessing hardware performance counters via PAPI accounts for substantially increased overhead, whereas sampling several PAPI counters needs only marginal extra overhead.

5 Trace Visualization with Vampir and VampirServer

The Vampir tool suite is a research project of the Technische Universität Dresden for analyzing the run-time behavior of parallel MPI/OpenMP software programs. It visualizes the program execution by means of event traces, gathered by monitoring software like KOJAK, TAU, or VampirTrace [16, 14, 20].

The visualization takes place after the monitored program has been completed, using data that has been captured during the program execution. Our approach called VampirServer introduces parallel performance data evaluation concepts which are implemented in a client-server framework. The server component can be installed on a segment of a parallel production environment. The corresponding clients can run on remote desktop computers to visualize the performance results graphically, see Fig. 2. The major advantages of this parallel, distributed approach are:

1. Performance data which tends to be bulky is kept where it was created.
2. Parallel processing significantly increases the scalability of the analysis process.
3. It works efficiently from arbitrary remote end-user platforms.
4. Very large trace files can be browsed and visualized interactively.

Visualization clients translate the condensed performance data into a variety of graphical representations providing developers with a good understanding of performance issues concerning their applications. This allows for quick focusing on appropriate levels of detail which allows the detection and explanation of various performance bottlenecks such as load imbalances and communication deficiencies.

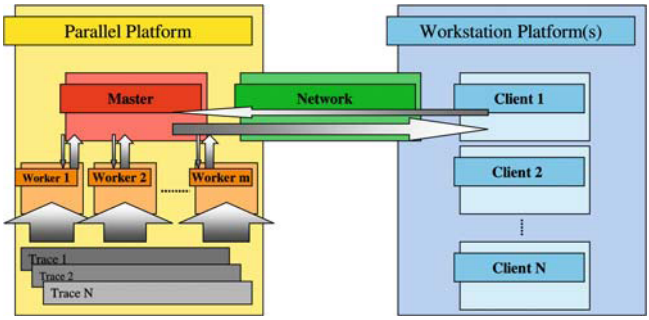


Fig. 2 An overview of the distributed software architecture for parallel performance analysis. The left image part provides the structure of the parallel analysis server component. On the right hand side N visualization components are depicted. Analysis and visualization components communicate over an encrypted socket communication channel on the Internet

5.1 Analysis Architecture and Scalability

During the evolution of the Vampir project, we identified three requirements with respect to current parallel computer platforms that typically cannot be fulfilled by classical sequential post mortem software analysis approaches:

1. exploit *distributed memory* for analysis tasks,
2. process both long (regarding time) and wide (regarding number of processes) program traces in *real-time*,
3. *limit the data* processed at the client end (workstation, laptop) to a volume that is independent of the amount of event trace data.

The previous section has provided a rough sketch of the analysis server's internal architecture, which will now be described in further detail. Figure 3 can be regarded as a close-up of the left part of the service architecture overview. On the right hand side we can see the MPI master process being responsible for the interaction with the clients and the control over the worker processes. On the left hand side *m* identical MPI worker processes are depicted.

Every single MPI worker process is equipped with one main thread handling MPI communication with the master and if required, with other MPI workers. The main thread is created once at the very beginning and keeps on running until the server is terminated. Depending on the number of clients to be served, every MPI process has a dynamically changing number of *n* session threads being responsible for the clients' requests. The communication between MPI processes is done with standard MPI constructs whereas the local threads communicate by means of shared buffers, synchronized by mutexes and conditional variables. This permits a low overhead during interactions between the mostly independent components.

Session threads can be subdivided into three different module categories as there are: analysis modules, event data base modules, and trace format modules. Starting from the bottom, trace format modules include parsers for the traditional Vampir

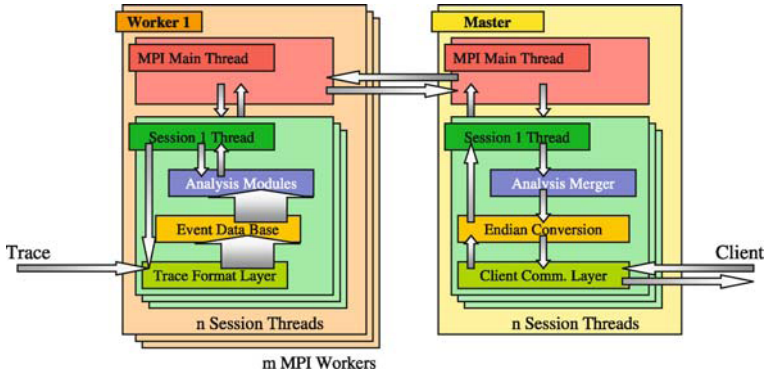


Fig. 3 The parallel analysis server in detail. The server consists of *m* worker processes (left) and one master process (right). The worker processes store performance data and handle analysis requests whereas the master process is responsible for merging and sending worker results to connected visualization clients

trace format (VTF), the newly designed scalable trace format (OTF) [11], and the EPILOG trace format (ELG) [17]. The modular approach allows to easily add other third party formats. The data base modules include storage objects for all supported event categories like functions, messages, performance metrics etc. The final module category provides the analysis capabilities of the server. This type of module performs its work on the data provided by the data base modules.

In contrast to the worker process described above the situation for the master process is slightly different. First of all, the layout with respect to its inherent threads is identical to the one applied on the worker processes. Similar to a worker process, the main thread is also responsible for doing all MPI communication with the workers. The session threads on the other hand have different tasks. They are responsible for merging analysis results received from several workers, converting the results to a platform independent format, and doing the communication with the clients like depicted on the right hand side of Fig. 3.

5.2 Zooming and Browsing in Timeline Displays

The most prominent displays provided by Vampir are the timeline displays. They show the sequence of recorded events on a horizontal time axis that can be zoomed to any detail level. Thus, timelines allow an in-depth analysis of the dynamic behavior of an application. There are several types of timeline displays:

The Global Timeline. It shows the processes and threads of the parallel program on the vertical axis. The program's state (i.e. the currently active function) is depicted by a horizontal bar, which is colored according to the associated function group. Point-to-point messages, global communication, as well as I/O operations

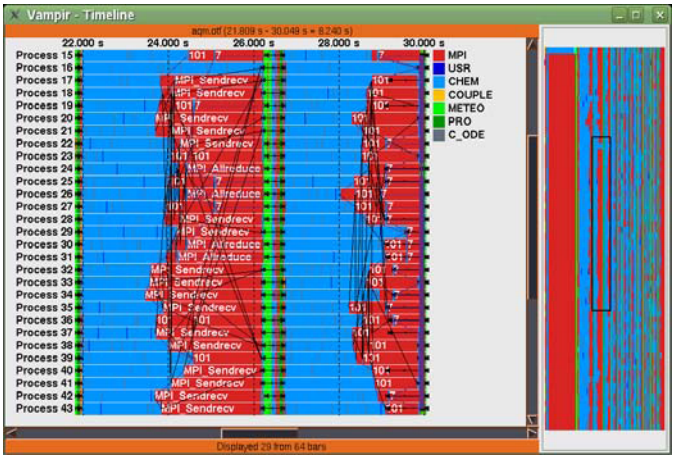


Fig. 4 Global Timeline display showing load imbalance. The calculation phase (CHEM) is not well balanced, which leads to high synchronization times of many processes (MPI)

are displayed by arrows. This allows a very detailed analysis of the parallel program flow, e.g. communication patterns, load imbalances, and I/O bottlenecks. An example of a load imbalance situation is shown in Fig. 4.

The Summary Timeline. It provides a stacked view of the number of processes involved at a given activity (i.e. function group) dynamically over time. In this display no individual processes can be identified, instead it allows a very concise high-level overview over many parallel processes/threads. By this means, phases with a high share of communication and synchronization can be easily pinpointed even in massively parallel programs.

The Counter Timeline. It displays selected counters for all processes next to each other, which is useful to locate anomalies indicating performance problems. The counters can be displayed as absolute values (e.g. allocated memory in bytes) or as rate (e.g. floating point operations per second).

The Process Timeline. In contrast to the other displays mentioned, it focuses on one process or thread. Here, the vertical axis shows the sequence of events in their respective call stack levels, allowing a detailed analysis of function calls. Performance counters and I/O events can be displayed aligned to the function calls. As an example, a complete Process Timeline for the WRF weather forecast model [1] is shown in Fig. 5. By zooming in, more and more details can be revealed up to the level of single function calls.

All displays are coupled in terms of the selected time interval. This means, zooming in one timeline display automatically updates all other displays (timelines and statistics) accordingly. Navigation in timelines is possible by scrollbars and by means of the thumbnail view, which shows a small overview of the complete timeline with zooming markers.

Fig. 5 Complete Process Timeline of a WRF process (right). By means of the call stack representation, one can clearly see the initialization phase and the iteration steps. It includes performance counters for memory allocation and floating point operations per second. A zoomed version shows individual I/O operations with very slow speed indicating a performance problem (left)

5.3 Statistics Displays

In addition to the timeline displays, Vampir provides a set of statistics displays, which show summarized information according to the currently selected time interval of the timeline view. This is the most interesting advantage over profiling data because it allows to show statistics specifically for selected time intervals.

Different statistics displays provide information about various aspects like execution times of functions or groups, the function call tree, point-to-point messages, collective communication, as well as I/O events.

The Summary Chart. It shows profile information about functions or groups. It allows to create profiles of parts of a complex scientific application, e.g. omitting initialization and finalization. Furthermore, this enables to compare different time intervals of the whole trace, e.g. different time steps of a numerical simulation. This is illustrated in Fig. 6, which shows the Summary Charts for two different time steps of the WRF weather code.

The Activity Chart. This display resembles the Summary Chart but focuses on a single process/thread only.

The Message Statistics Display. To analyze point-to-point communication of MPI applications, the **Message Statistics** provide a sender-receiver matrix of statistics. It can display different kinds of information, like message count, and lengths, speed and duration of messages. Again, this display may become very large with a large number of processes. Therefore, a thumbnail provides an overview and a navigational aid.

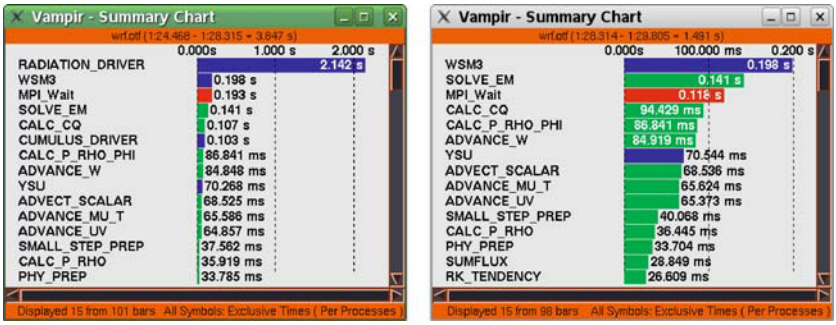


Fig. 6 Summary Charts of two different time steps of the WRF model showing exclusive execution times of the functions. The time step selected in the left window includes radiation calculation, which is very time consuming

Figure 7 shows the Message Statistics of the a WRF model displaying the average message transfer rate. Slow transfer rates may not only be caused by an actual slow message transfer, but also due to delays if one of the communication partner needs to wait for the other to start sending or receiving.

Further types of statistics displays include the **Collective Communication Statistics**, the **I/O Event Statistics**, the **Process Profile**, the **Call Tree**, and the **Message Profile**. Refer to [7] for a complete list and comprehensive description.

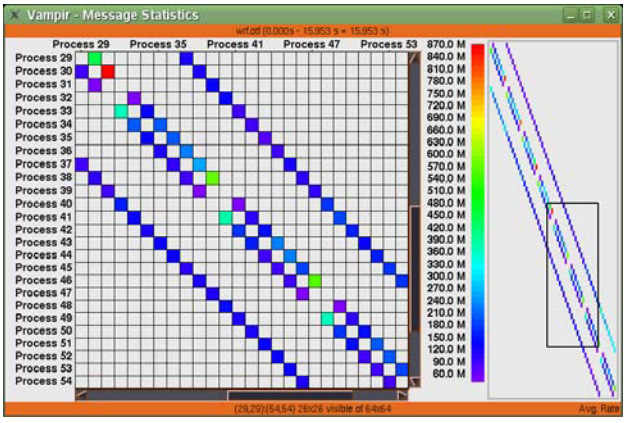


Fig. 7 Message Statistics of the WRF model showing a typical pattern of a parallel CFD code. in this screenshot, the average message transfer rates are displayed

6 Related Work

There are many other academic and commercial trace-based performance analysis tools with focus on HPC and parallel computing. The Paraver tools from Technical University of Catalonia in Barcelona, Spain [8] and Trace Collector and Trace Analyzer tools from Intel [3, 2] use very similar trace analysis methods with respect to instrumentation, event recording and visual displays.

The TAU toolkit from University of Oregon, US uses similar instrumentation techniques, but focuses on elaborate profiling (even though it supports trace analysis as well) and provides statistical performance evaluation and visualization.

The tools from the KOJAK project as well as from the successor project Scalasca from the Jülich Supercomputing Centre (JSC) allow automatic detection of performance problems for parallel and massively parallel applications [16, 5].

7 Conclusions and Future Work

VampirTrace and Vampir are robust event trace analysis tools for parallel applications that work on many UNIX-like platforms. Both support analysis of a number of important performance properties like function calls, hardware performance counters, communication, I/O behavior and memory allocation.

In the near future, VampirTrace is going to be integrated into Open MPI, a well known collaborative open source MPI implementation [19]. The next major release Open MPI 1.3 which is expected in the first half of 2008 will include VampirTrace by default. In addition to the standard MPI compiler wrappers `mpicc`, `mpicxx`, `mpif90`, etc. it will provide alternative versions `mpicc-vt`, `mpicxx-vt`, `mpif90-vt` that perform VampirTrace instrumentation.

For Vampir a port to Microsoft Windows is underway which is expected to be available as a beta version in 2009.

Current research focuses on elimination of redundancy in traces and pattern detection in large event data sets via Complete Call Graphs (CCG) [12]. This will allow more efficient in-memory storage during analysis.

In long, iterative as well as SIMD-style (single instruction multiple data) parallel applications there is a high degree of repetition in the program execution. It leads to repeated event sequences with *identical* or *similar* timing and behavior. It is possible to replace such redundant entities with references to a single instance which leads to reduced memory consumption as well as to more efficient trace analysis.

This approach is specially designed for event tracing data. In particular, it needs to guarantee a suitable definition of *similar* behavior such that insignificantly small deviations are ignored (e.g. few ticks in timing) but important differences are preserved (e.g. process/thread IDs or MPI ranks).

Based on compressed in-memory data structures, detection of repetition patterns can be done very efficiently. Furthermore, it is easy to differentiate between regular (good) behavior and irregular outliers (bad) for repeated sub-sequences of events.

References

1. The Weather research and forcasting system WRF. http://wrf-model.org
2. Corp., I.: Intel (R) Trace Analyzer 7.1 Reference Guide (2007). http://www.intel.com/ , document number 318120
3. Corp., I.: Intel (R) Trace Collector 7.1 User's Guide (2007). http://www.intel.com/, document number 318119
4. Fenlason, J., Stallman, R.: GNU gprof
5. Geimer, M., Kuhlmann, B., Pulatova, F., Wolf, F., Wylie, B.J.N.: Scalable Collation and Presentation of Call-Path Profile Data with CUBE. In: Parallel Computing: Architectures, Algorithms and Applications (Proceedings of the International Conference ParCo 2007), pp. 645–652. Jülich/Aachen, Germany (2007)
6. Graham, S.L., Kessler, P.B., McKusick, M.K.: gprof: a Call Graph Execution Profiler. In: SIGPLAN Symposium on Compiler Construction, pp. 120–126 (1982). URL citeseer.ist.psu.edu/graham82gprof.html
7. GWT TU Dresden mbH: VampirServer 1.8 User Manual (2008). http://www.vampir./eu/
8. Jost, G., Jin, H., Labarta, J., Gimenez, J.: Interfacing Computer Aided Parallelization and Performance Analysis. In: Proceedings of the International Conference on Computational Science (ICCS) (2003)
9. Jurenz, M.: VampirTrace Software and Documentation. ZIH, TU Dresden (2006). http://www.tu-dresden.de/zih/vampirtrace/
10. Jurenz, M., Brendel, R., Knüpfer, A., Müller, M.S., Nagel, W.E.: Memory Allocation Tracing with VampirTrace. In: International Conference on Computational Science (2), pp. 839–846 (2007)
11. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the Open Trace Format (OTF). In: Proc. of ICCS 2006: 6'th Intl. Conference on Computational Science, Springer LNCS 3992, pp. 526 – 533. Reading, UK (2006)
12. Knüpfer, A., Nagel, W.E.: Compressible Memory Data Structures for Event-Based Trace Analysis. Future Generation Computer Systems **22**(3), 359–368 (2006)
13. Malony, A.D., Mohr, B., Wolf, F., Shende, S.: Design and Prototype of a Performance Tool Interface for OpenMP. The Journal of Supercomputing **Vol. 23**, 105–128 (2002)
14. Malony, A.D., Shende, S., Bell, R., Li, K., Li, L., Trebon, N.: Advances in the TAU performance system pp. 129–144 (2004)
15. Mickler, H., Kluge, M., Knüpfer, A., Müller, M.S., Nagel, W.E.: Tracing Application I/O Calls with VampirTrace. In: Euro-Par '08: Proc. from the 14th Intl. Euro-Par Conference on Parallel Processing (2008). (Submitted for publication)
16. Mohr, B., Wolf, F.: KOJAK: A Tool Set for Automatic Performance Analysis of Parallel Applications. Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003) pp. 1301–1304 (2003)
17. Mohr, B., Wolf, F.: EPILOG Binary Trace-Data Format. Tech. Rep. FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, University of Tennessee (2004)
18. Müller, M., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In: C. Bischof, M. Bücker, P. Gibbon, G. Joubert, T. Lippert, B. Mohr, F. Peters (eds.) Parallel Computing: Architectures, Algorithms and Applications, Proc. of ParCo 2007, vol. 38, pp. 637–644. NIC-Series (2007)
19. Open MPI website. http://www.open-mpi.org/
20. Shende, S., Malony, A.D.: The Tau Parallel Performance System. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006). DOI 10.1177/1094342006064482

Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications

Felix Wolf[1,2], Brian J. N. Wylie[1], Erika Ábrahám[1], Daniel Becker[1,2], Wolfgang Frings[1], Karl Fürlinger[3], Markus Geimer[1], Marc-André Hermanns[1], Bernd Mohr[1], Shirley Moore[3], Matthias Pfeifer[1,2], and Zoltán Szebenyi[1,2]

Abstract SCALASCA is a performance toolset that has been specifically designed to analyze parallel application behavior on large-scale systems, but is also well-suited for small- and medium-scale HPC platforms. SCALASCA offers an incremental performance-analysis process that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. A distinctive feature of SCALASCA is its ability to identify wait states even for very large processor counts. The current version supports the MPI, OpenMP and hybrid programming constructs most widely used in highly-scalable HPC applications.

1 Introduction

Supercomputing is a key technology pillar of modern science and engineering, indispensable to solve critical problems of high complexity. World-wide efforts to build machines with performance levels in the petaflops range acknowledge that the requirements of many key applications can only be met by the most advanced custom-designed large-scale computer systems. However, as a prerequisite for their productive use, the HPC community needs powerful and robust performance-analysis tools that make the optimization of parallel applications both more effective and more efficient. Such tools not only help improve the scalability characteristics of scientific codes and thus expand their potential, but also allow domain experts to

1. Jülich Supercomputing Centre, Forschungszentrum Jülich, Germany
{f.wolf,b.wylie,e.abraham,d.becker,w.frings,m.geimer,
m.a.hermanns,b.mohr,m.pfeifer,z.szebenyi}@fz-juelich.de

2. Department of Computer Science and Aachen Institute for Advanced Study in Computational Engineering Science, RWTH Aachen University, Germany

3. Innovative Computing Laboratory, University of Tennessee, USA
{karl,shirley}@cs.utk.edu

concentrate on the science underneath rather than to spend a major fraction of their time tuning their application for a particular machine.

As the current trend in microprocessor development continues, this need will become even stronger in the future. Facing increasing power dissipation and with little instruction-level parallelism left to exploit, computer architects are realizing further performance gains by using larger numbers of moderately fast processor cores rather than by further increasing the speed of uni-processors. As a consequence, supercomputer applications are being required to harness much higher degrees of parallelism in order to satisfy their growing demand for computing power. With an exponentially rising number of cores, the often substantial gap between peak performance and the performance actually sustained by production codes [6] is expected to widen even further. Finally, increased concurrency levels place higher scalability demands not only on applications but also on parallel programming tools [10]. When applied to larger numbers of processes, familiar tools often cease to work satisfactorily (e.g., due to escalating memory requirements, failing displays, or limited I/O bandwidth).

Developed at the Jülich Supercomputing Centre in cooperation with the University of Tennessee, SCALASCA is a performance-analysis toolset that has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but is also well-suited for small- and medium-scale HPC platforms. SCALASCA supports an incremental performance-analysis process that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. A distinctive feature of SCALASCA is its ability to identify wait states that occur, for example, as a result of unevenly distributed workloads. Especially when trying to scale communication-intensive applications to large processor counts, such wait states can present severe challenges to achieving good performance. Compared to its predecessor KOJAK [11], SCALASCA can detect such wait states even in very large configurations of processes using a novel parallel trace-analysis scheme [3].

In this article, we give an overview of SCALASCA and show its capabilities for diagnosing performance problems in large-scale parallel applications. First, we review the SCALASCA analysis process and discuss basic usage. After presenting the SCALASCA instrumentation and measurement systems in Section 3, Section 4 explains how its trace analysis can efficiently detect wait states in communication and synchronization operations even in very large configurations of processes, before we demonstrate how execution performance analysis reports can be interactively explored in Section 5. Finally, in Section 6, we outline our development goals for the coming years.

2 Overview

The current version of SCALASCA supports measurement and analysis of the MPI, OpenMP and hybrid programming constructs most widely used in highly-scalable HPC applications written in C/C++ and Fortran on a wide range of current HPC

platforms. Usage is primarily via the `scalasca` command with appropriate action flags, and is identical for a 64-way OpenMP application on a single UltraSPARC-T2 processor or a 64k hybrid OpenMP/MPI application on a Blue Gene/P.

Figure 1 shows the basic analysis workflow supported by SCALASCA. Before any performance data can be collected, the target application must be *instrumented*, that is, it must be modified to record performance-relevant events whenever they occur. On most systems, this can be done completely automatically using compiler support; on other systems a mix of manual and automatic instrumentation mechanisms is offered. When running the instrumented code on the parallel machine, the user can choose between generating a summary report (aka profile) with aggregate performance metrics for individual function call paths, or generating event traces recording individual runtime events from which a profile or time-line visualization can later be produced. The first option is useful to obtain an overview of the performance behavior and also to optimize the instrumentation for later trace generation. Since traces tend to become very large, this step is usually recommended before choosing the second option. When tracing is enabled, each process generates a trace file containing records for all its process-local events. After program termination, SCALASCA loads the trace files into main memory and analyzes them in parallel using as many CPUs as have been used for the target application itself. During the analysis, SCALASCA searches for characteristic patterns indicating wait states and related performance properties, classifies detected instances by category and quantifies their significance. The result is a pattern-analysis report similar in structure to the summary report but enriched with higher-level communication and synchronization inefficiency metrics. Both summary and pattern reports contain performance metrics for every function call-path and system resource which can be interactively explored in a graphical report explorer (Fig. 3). As an alternative to the automatic search, the event traces can be converted and investigated using third-party trace browsers such as Paraver [4, 7] or VAMPIR [5, 9], taking advantage of their powerful time-line visualizations and rich statistical functionality.

3 Instrumentation and Measurement

SCALASCA offers analyses based on two different types of performance data: (i) aggregated statistical summaries and (ii) event traces. By showing which process consumes how much time in which call-path, the summary report provides a useful overview of an application's performance behavior. Because it aggregates the collected metrics across the entire execution, the amount of data is largely independent of the program duration. This is why runtime summarization is the first choice for very long-running programs working on realistic input data sets and models. The summary metrics measured with SCALASCA include wall-clock time, the number of times a call-path has been visited, message counts, bytes transferred, and a rich choice of hardware counters available via the PAPI library [2].

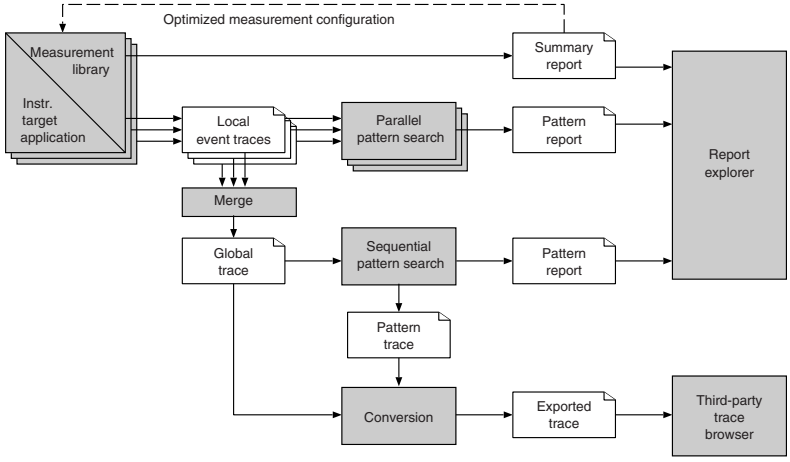


Fig. 1 SCALASCA's performance analysis workflow

In contrast, event traces allow the in-depth study of parallel program behavior. Tracing is especially effective for observing the interactions between different processes or threads that occur during communication or synchronization operations and to analyze the way concurrent activities influence each other's performance. When an application is traced, SCALASCA records individual performance-relevant events with timestamps and writes them to a trace file (one per process) to be analyzed in a subsequent step.

To effectively monitor program execution, SCALASCA intercepts runtime events critical to communication and computation activities. These events include entering and leaving functions or other code regions as well as sending and receiving point-to-point messages or participation in collective communication. Whereas the communication-related event types are crucial to study the interactions among different processes and to identify wait states, function entries and exits are needed to understand the computational requirements and the context in which the most demanding communication operations occur.

The application must be instrumented to provide notification of these events during measurement, using function calls inserted at specific important points ("events") which call into the SCALASCA measurement library. Just linking the application with the measurement library already ensures that all events related to MPI operations are properly captured. For OpenMP, a source preprocessor is used which automatically instruments directives and pragmas for parallel regions, etc., and many compilers are capable of adding instrumentation to every function or routine entry and exit. Finally, programmers can manually add their own custom instrumentation annotations in the source code for important regions (such as phases or loops, or functions when this is not done automatically by the compiler): these annotations are in the form of pragmas or macros which are ignored when instrumentation is not configured.

Instrumentation configuration and processing of source files are achieved by prefixing the SCALASCA instrumenter to selected compilation commands and the final link command, without requiring other changes to optimization levels or the build process.

```
# scalasca -instrument <compile-or-link-command>
% scalasca -instrument mpicc -c foo.c
% scalasca -instrument f90 -o bar -OpenMP bar.F
% scalasca -instrument mpif90 -o foobar -OpenMP foo.o bar.F
```

A simple means to be able to conveniently instrument an entire application, is to add a 'preparer' prefix to compile and link commands in its Makefile(s), which is undefined by default and results in a regular uninstrumented build, or when the preparer is set to the SCALASCA instrumenter then an instrumented build is produced.

```
PREP =
MPICC = $(PREP) mpicc
MPIFC = $(PREP) mpif90
foobar: bar.F foo.o
       $(MPIFC) -o $@ -OpenMP foo.o bar.F

% make PREP="scalasca -instrument"
```

The SCALASCA measurement system that gets linked with instrumented application executables can be configured to allow runtime summaries and/or event traces to be collected, along with optional hardware counter metrics. A unique experiment archive is created to contain all of the measurement and analysis artifacts, including configuration information, log files and analysis reports. When event traces are collected, they are also stored in the experiment archive to avoid accidental corruption by simultaneous or subsequent measurements.

Measurements are collected and analyzed under the control of a nexus which automatically configures the parallel trace analyzer with the same number of processes as used for measurement. This allows SCALASCA analysis to be specified as a command prefixed to the application execution command-line, whether part of a batch script or run interactively.

```
# scalasca -analyze <application-launch-command>
% scalasca -analyze mpiexec -np 65536 foo arglist
Scalasca runtime summarization experiment ./epik_foo_65536_sum
% OMP_NUM_THREADS=64 scalasca -analyze bar arglist
Scalasca runtime summarization experiment ./epik_bar_0x64_sum %
OMP_NUM_THREADS=4 scalasca -analyze mpiexec -np 512 foobar
Scalasca runtime summarization experiment ./epik_foobar_512x4_sum
```

Although collection of runtime summarization experiments is the default, addition of the `-t` flag configures trace collection and automatic analysis (without the need for instrumentation re-configuration).

```
% OMP_NUM_THREADS=4 scalasca -analyze -t mpiexec -np 512 foobar
Scalasca trace analysis experiment ./epik_foobar_512x4_trace
```

Instrumented functions which are executed frequently, while only performing a small amount of work each time they are called, have an undesirable impact on measurement. The overhead of measurement for such functions is large compared to the execution time of the (uninstrumented) function, resulting in measurement dilation, while recording such events requires significant space and analysis takes longer with relatively little improvement in quality. This is especially important for event traces whose size is proportional to the total number of events recorded. For this reason, SCALASCA offers various mechanisms to exclude certain functions from measurement. Before writing a trace file, the instrumentation should therefore be optimized based on a visit-count summary obtained during an earlier run.

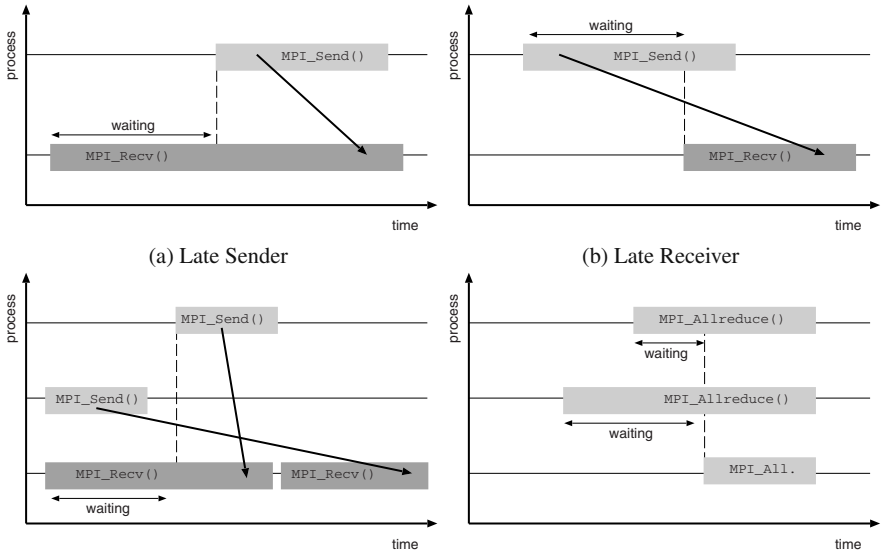


Fig. 2 Examples for patterns of inefficient behavior. Note that the combination of MPI functions used in each of these examples represents just one possible case

4 Trace Analysis

In message-passing applications, processes often require access to data provided by remote processes, making the progress of a receiving process dependent upon the progress of a sending process. If a rendezvous protocol is used, this relationship also applies in the opposite direction. Collective synchronization is similar in that its completion requires each participating process to have reached a certain point. As a consequence, a significant fraction of the time spent in communication and

synchronization routines can often be attributed to wait states that occur when processes fail to reach implicit or explicit synchronization points in a timely manner, for example, as a result of an unevenly distributed workload. Especially when trying to scale communication-intensive applications to large process counts, such wait states can present severe challenges to achieving good performance. As a first step in reducing the impact of wait states, SCALASCA provides a diagnostic method that allows their localization, classification, and quantification. Because wait states cause temporal displacements between program events occurring on different processes, their identification can be accomplished by searching event traces for characteristic patterns. A subset of the patterns supported by SCALASCA is depicted in Fig. 2.

As the first example of a typical wait state, consider the so-called *Late Sender* pattern (Fig. 2(a)). Here, a receive operation is entered by one process before the corresponding send operation has been started by the other. The time lost waiting due to this situation is at least the time difference between the two function invocations. In contrast, the *Late Receiver* pattern (Fig. 2(b)) describes the inverse situation, where a sender is blocked while waiting for the receiver when a rendezvous protocol is used (e.g., to transfer a large message). The *Late Sender / Wrong Order* pattern (Fig. 2(c)) is more complex than the previous two. Here, a receiver waits for a message, although an earlier message is ready to be received by the same destination process (i.e., message receipt in wrong order). Finally, the *Wait at N×N* pattern (Fig. 2(d)) quantifies the waiting time due to the inherent synchronization in collective n-to-n operations, such as `MPI_Allreduce`.

To accomplish the search is a scalable way, SCALASCA exploits both distributed memory and parallel processing capabilities available on the target system. Instead of sequentially analyzing a single global trace file, as done by its predecessor tool KOJAK, SCALASCA analyzes separate process-local trace files in parallel by *replaying* the original communication on as many CPUs as have been used to execute the target application itself. During the search process, SCALASCA classifies detected pattern instances by category and quantifies their significance for every program phase and system resource involved. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of application processes, SCALASCA has completed pattern searches even at the previously intractable scale of over 22,000 processes. Additionally, to allow accurate trace analyses on systems without globally synchronized clocks such as most PC clusters the trace analyzer provides the ability to synchronize inaccurate timestamps postmortem using the same scalable replay mechanism [1].

OpenMP Support and Pattern Traces

In addition to the scalable MPI trace analysis, sequential trace analysis (Fig. 1) is also provided for OpenMP and MPI one-sided RMA operations. This sequential analysis is currently the default for pure OpenMP measurements, and can be specified for an augmented analysis of MPI and hybrid measurements when desired. For large measurements, however, the additional storage space and serial analysis time required

can be prohibitive, unless very targeted instrumentation is configured or the problem size is reduced (e.g., to only a few timesteps or iterations). Other options include visual analysis using third-party trace browsers, such as Paraver and VAMPIR and the generation of pattern traces.

The first step to access these features consists of merging the local trace files generated by SCALASCA into a single global trace file. The resulting global trace file can then be searched for MPI and/or OpenMP patterns or converted and loaded into Paraver or VAMPIR. A third option was motivated by the fact that the pattern search method accumulates the severities of all of the pattern instances found to inform about the overall performance penalty. However, the temporal and spatial relationships between individual pattern instances are lost, although these relationships can be essential to understand the detailed circumstances of a performance problem. These relationships can now be retained by writing a second event trace with events delimiting individual pattern occurrences. Guided by the summary pattern report, this synthetic pattern trace can be interactively analyzed leveraging the powerful functionality of the aforementioned trace browsers.

5 Understanding Performance Behavior

After SCALASCA analysis is completed, the experiment archive may contain a summary report generated immediately at measurement completion and/or trace-analysis report(s) generated after searching event traces. These profiles have the same structure and can be viewed and manipulated using the same set of commands.

```
# scalasca -examine <experiment-archive>
% scalasca -examine epik_foobar_512x4_trace
```

Whereas a summary report includes metrics, such as time, visit counts, message statistics or hardware counters, a trace-analysis report also accounts for the times lost in different wait states. Both types of reports are stored as a three-dimensional array with the dimensions metric, call path, and system resource (e.g., process or thread). Because of the cubic structure, the corresponding file format is called CUBE. For every metric included, a CUBE report stores the aggregated value for each combination of call-path and process or thread. Motivated by the need to represent performance behavior on different levels of granularity as well as to express natural hierarchical relationships among metrics, program, or system resources, each dimension is organized in a hierarchy.

The SCALASCA analysis report explorer (Fig. 3) provides the ability to interactively browse through this three-dimensional performance data space in a convenient way. Its design emphasizes simplicity by combining a small number of orthogonal features with a limited set of user actions. Each dimension of the data space (metric, call-path, and system resource) can be shown using tree displays and allows the user to interactively explore the values of all the data points. Since the data space is large, views representing only a subspace can be selected and combined

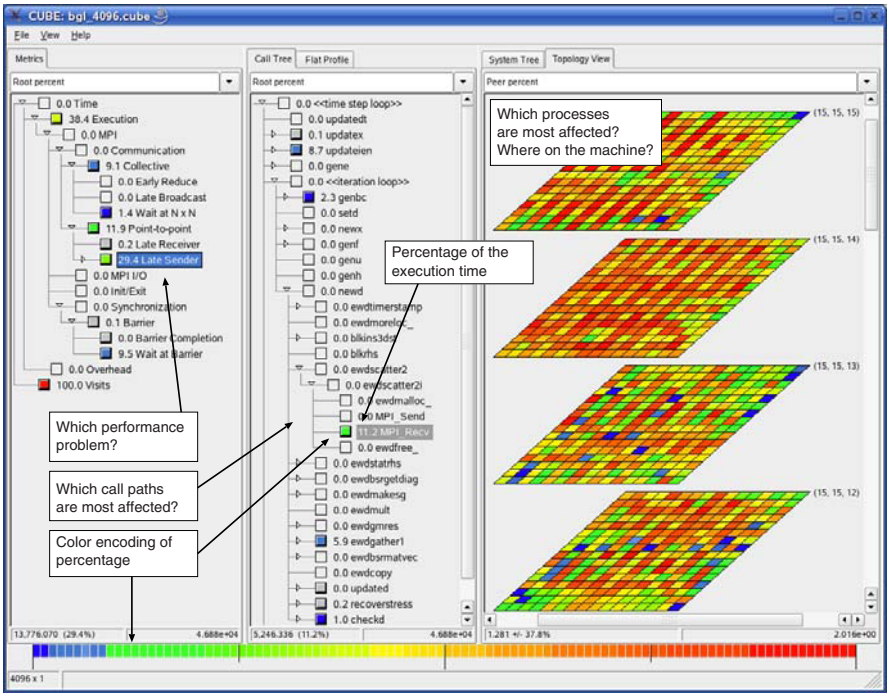


Fig. 3 The trace analysis report displayed in the report explorer indicates that 29.4% of the total time in the annotated region `<<timestep loop>>` is spent waiting due to *Late Sender* situations (left pane). The call tree (middle pane) shows that more than one-third of the waiting time is concentrated in one call-path, with its waiting time unevenly distributed across the visible section of the machine topology (right pane)

with aggregation mechanisms that control the level of detail. Two types of actions can be performed: selecting a node or expanding/collapsing a node. Whereas the first action defines a "slice" or "column" of the data space, the latter exposes/hides sub-hierarchies of the different dimensions. To help identify combinations with a high value more quickly, all values are not only shown numerically but also color-coded. To facilitate the analysis of runs on many processors, the explorer provides a scalable two- or three-dimensional Cartesian grid display to visualize physical or virtual process topologies which were recorded with measurements. The topological display is offered as an alternative to a standard tree hierarchy of machine, compute nodes, processes and threads.

With a set of command-line tools [8], CUBE reports can be combined or manipulated to allow comparisons or aggregations of different reports or to focus the analysis on specific parts of a report. Specifically, multiple reports can be averaged or merged, the difference between two reports calculated, or a new report generated after pruning specified call-trees and/or specifying a call-tree node as a new root. The latter can be particularly useful for eliminating uninteresting phases (e.g., initialization) and focusing the analysis on a selected part of the execution. These

utilities each generate new CUBE-format reports as output that can be loaded into the explorer like the original reports that were used as input.

6 Outlook

Future enhancements will aim at both further improving the functionality and scalability of the SCALASCA toolset. Whereas automatic MPI analysis has been demonstrated at very large scales, runtime summaries currently only include measurements for the OpenMP master thread, and OpenMP trace analysis is currently done serially: for hybrid applications, scalable MPI trace analysis is the default and serial OpenMP analysis is offered as an additional option. Most standard-conforming HPC applications should be measurable, however, there is no recording or analysis of MPI I/O, experimental analysis of MPI one-sided RMA operations is currently only done by the serial trace analyzer, and automatic trace analysis of OpenMP applications using dynamic, nested and guarded worksharing constructs is not yet possible.

While the current parallel trace analysis mechanism is already a very powerful instrument in terms of the number of application processes it supports, we are working on optimized data management operations and workflows that will allow us to master even larger configurations. Restrictions and inefficiencies imposed by the current CUBE-file format and data model are also being addressed to allow non-aggregatable metrics (such as rates) to be stored and accessed without the need to process and aggregate values from the entire report.

Although parallel simulations are often iterative in nature, individual iterations can differ in their performance characteristics. Another major focus of our research is therefore to study the temporal evolution of the performance behavior as a computation progresses. Our general approach is to first observe the behavior on a coarse-grained level and then to successively refine the measurement focus as new performance knowledge becomes available. Using a more flexible measurement control, we are also striving to offer more targeted trace collection mechanisms, reducing memory and disk space requirements while retaining the value of trace-based in-depth analysis.

Finally, the symptoms of a performance bottleneck may appear much later than the event causing it, on a different processor, or both. For this reason, we are currently looking for ways to establish causal connections among different pattern instances found in traces and related phenomena such as load imbalance because we believe that understanding such links can prove essential for more effective scaling strategies. First experiments with a trace-based simulator that verifies corresponding hypotheses by replaying modified traces in real time on the target system proved encouraging.

For more information on SCALASCA refer to the website www.scalasca.org.

Acknowledgements This work has been supported by the Helmholtz Association under Grants No. VNG-118 and No. VH-VI-228 ('VI-HPS') and by the Federal Ministry for Research and Education (BMBF) under Grant No. 01IS07005C ('ParMA').

References

1. Becker, D., Rabenseifner, R., Wolf, F.: Timestamp synchronization for event traces of large-scale message-passing applications. In: Proc. of the 14th European Parallel Virtual Machine and Message Passing Interface Conference (EuroPVM/MPI), *Lecture Notes in Computer Science*, vol. 4757, pp. 315–325. Springer, Paris, France (2006)
2. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. International Journal of High Performance Computing Applications **14**(3), 189–204 (2000)
3. Geimer, M., Wolf, F., Wylie, B., Mohr, B.: Scalable parallel trace-based performance analysis. In: Proc. of the 13th European Parallel Virtual Machine and Message Passing Interface Conference (EuroPVM/MPI), *Lecture Notes in Computer Science*, vol. 4192, pp. 303–312. Springer, Bonn, Germany (2006)
4. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: DiP : A parallel program development environment. In: Proc. of the 2nd International Euro-Par Conference, pp. 665–674. Springer, Lyon, France (1996)
5. Nagel, W., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer **63, XII**(1), 69–80 (1996)
6. Oliker, L., Canning, A., Carter, J., Iancu, C., Lijewski, M., Kamil, S., Shalf, J., Shan, H., Strohmaier, E., Ethier, S., Goodale, T.: Scientific application performance on candidate petascale platforms. In: Proc. of the International Parallel & Distributed Processing Symposium (IPDPS). Long Beach, CA (2007)
7. Paraver: http://www.cepba.upc.es/paraver/
8. Song, F., Wolf, F., Bhatia, N., Dongarra, J., Moore, S.: An algebra for cross-experiment performance analysis. In: Proc. of the International Conference on Parallel Processing (ICPP), pp. 63–72. IEEE Society, Montreal, Canada (2004)
9. VAMPIR: http://www.vampir.eu/
10. Vanter, M.V.D., Post, D., Zosel, M.: HPC needs a tool strategy. In: Proc. of the 2nd International Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS) (2005)
11. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture **49**(10-11), 421–439 (2003)

Evolution of a Parallel Performance System

Allen D. Malony, Sameer Shende, Alan Morris, Scott Biersdorff, Wyatt Spear, Kevin Huck, and Aroon Nataraj

Abstract The TAU Performance System® is an integrated suite of tools for instrumentation, measurement, and analysis of parallel programs targeting large-scale, high-performance computing (HPC) platforms. Representing over fifteen calendar years and fifty person years of research and development effort, TAU's driving concerns have been portability, flexibility, interoperability, and scalability. The result is a performance system which has evolved into a leading framework for parallel performance evaluation and problem solving. This paper presents the current state of TAU, overviews the design and function of TAU's main features, discusses best practices of TAU use, and outlines future development.

1 Introduction

Scalable parallel systems have always evolved together with the tools used to observe, understand, and optimize their performance. Next-generation parallel computing environments are guided to a significant degree by what is known about application performance on current machines and how performance factors might be influenced by technological innovations. State-of-the-art performance tools play an important role in helping us understand application performance and allowing us to focus our attention on future performance concerns. Therefore, performance technology must keep pace with the growing complexity of next-generation parallel platforms if they are to contribute to the present and future promises of high-end parallel computing (HEC). We will need a robust performance frameworks that can provide both flexible and portable empirical performance observation capabilities at all levels of a system. In short, mapping low-level behavior to high-level performance abstractions to be understood within a parallel programming paradigm. It will be a challenge to develop such a *parallel performance systems*. It will need to respond to

Performance Research Lab, University of Oregon, Eugene, OR, e-mail: {malony,sameer,amorris,scottb,wspear,khuck,anataraj}@cs.uoregon.edu

new requirements for performance evaluation, problem diagnosis, and optimization as parallel computing technology extends it reach to high-end machines of massive scale.

There are four design objectives for parallel performance systems that will allow then to keep pace with HEC advancement. First, a performance system should be *flexible*. It should give maximum opportunity for configuring performance experiments that are of interest. In general, flexibility in empirical performance evaluation implies freedom in experiment design and in the selection and control of experiment mechanisms. Using tools that otherwise limit the type and structure of performance mechanisms will restrict scope of the application that can be evaluated. Second, a performance system should be *portable*, to allow consistent cross-platform performance problem solving. To achieve Portability one should look for common abstractions in performance methods and how these techniques can be reused across different computing environments. Lack of a portable performance evaluation environment forces users to adopt different techniques on different systems, even for common performance analysis tasks. A third objective is *integration*. An integrated performance system, by using explicit interfaces and common data formats, can work together as a unified framework. Finally, an *interoperable* parallel performance system allows performance technology from other tool suites to be leveraged for extended capability.

The TAU Performance System [39, 22, 38, 44] is the product of fifteen years of development to create a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications. The success of the TAU project represents the combined efforts of researchers at the University of Oregon and colleagues at the Research Centre Jülich and Los Alamos National Laboratory. This paper gives an overview of TAU's system architecture and current suite of tools, as well as discussing the best practices when using TAUĖxamples will be drawn from recent work highlighting some of TAU's new features. Given TAU's continued evolution as a parallel performance system, the paper will also provide a forecast of what is on the horizon.

2 TAU Performance System Design and Architecture

TAU is designed as a tool framework, wherein tool components and modules integrate and coordinate their operations using well-defined interfaces and data formats. The TAU framework architecture, shown in Fig. 1, is organized into three primary layers – *instrumentation*, *measurement*, and *analysis* – within each layer multiple modules are available and can be configured in a flexible manner by the user. The following sections discuss the layers in more detail, but now let us discuss the overall design decisions that governed TAU's development.

TAU is a performance systems based on *direct performance observation*, whereby execution *actions* of interest are exposed as *events* to the performance system

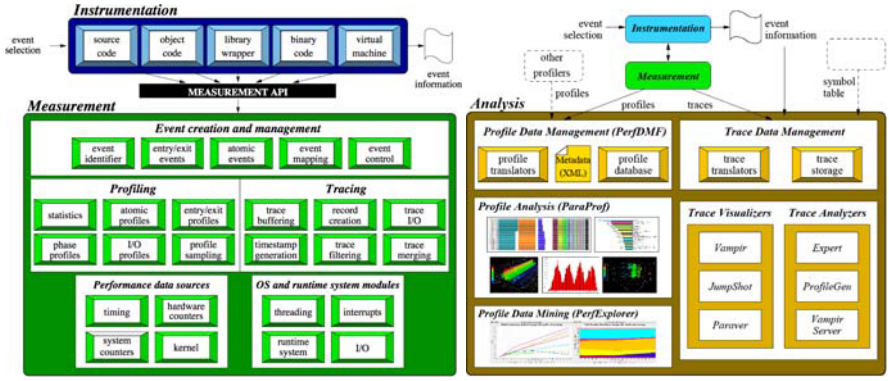


Fig. 1 TAU framework architecture

through direct insertion of instrumentation in the application, library, or system code, at locations where these actions arise. In general, these actions reflect some execution state, most commonly a result of a code location being reached (e.g., entry in a subroutine). However, it could also include a change in data. The key point is that the observation mechanism is direct. Generated events are made visible to the performance system in this way and contain implicit meta information about the associated action. Thus, for any *performance experiment* using direct observation, the *performance events of interest* must be decided beforehand and necessary instrumentation inserted.

The role of the instrumentation layer in TAU is to provide support for generating events at the appropriate execution points in the code. Since performance events of interest can be found at different places in the code, TAU provides a range of instrumentation capabilities to gather events from all these locations. The instrumentation function is simple: to insert code that calls the TAU measurement system when a specific action occurs and to gather the appropriate performance data.

TAU supports two classes of events: *atomic* events and *interval* events. An atomic event denotes a single action. When it occurs, the measurement system has the opportunity to obtain the performance data associated with that action at that time. In contrast, a interval event is really a pair of events: *begin* and *end*. The measurement system uses performance data obtained from each event to calculate a combined performance result (e.g., the time spent in a subroutine from entry (beginning of the interval) to exit (end of the interval)).

In addition to performance events, direct performance observation methods must obtain information about parallel execution context in order to interpret the parallel performance data. TAU was originally conceived [39] with the goal of supporting alternative models of parallel computation, from shared memory multi-threading to distributed memory message passing to mixed-mode parallelism. Instead of limiting attention to a sub-class of computation models, thereby reducing tool coverage, TAU defines an abstract computation model for parallel systems that captures general architecture and software features and can be mapped straightforwardly to exist-

ing complex system types [38]. The practical consequence is that TAU tags parallel performance measurements with *node:context:thread* information. In the model, a *node* is defined as a physically distinct machine with one or more processors sharing a physical memory system (i.e., a shared memory multiprocessor (SMP)). A *context* is a distinct virtual address space within a node providing shared memory support for parallel software execution. Multiple *threads* of execution can be active within a context.

Given performance events and their parallel context, TAU supports the two dominant methods of measurement for direct performance observation: *profiling* and *tracing*. Profiling methods compute performance statistics at runtime based on measurements of atomic or interval events. Tracing, on the other hand, records the measurement information for each event (including when it occurred) in a file for future analysis. In profiling, the number of recorded events is fixed, whereas tracing will generate a record for every event occurrence. The TAU performance system supports both parallel profile and parallel trace analysis with profiling tools developed internally as well as tracing tools incorporated from other groups.

Overall, TAU's design has proven to be robust, sound, and highly adaptable to each generations of parallel systems. In recent years, we have extended the TAU performance system architecture to support kernel-level performance integration [29], performance monitoring [32, 30], and collaboration through the *T*AU *Portal* [45]. The extensions have been moderate, mainly in the development of new interfaces for performance data access and reporting.

In the next sections, TAU's instrumentation, measurement, and analysis technology is described in more detail. Description of basic TAU usage is also given as brief introduction to its more advanced capabilities.

3 TAU Instrumentation

Instrumentation for direct performance observation involves inserting code (a.k.a. *probes*) to make performance events visible to the measurement substrate and to provide event control. From TAU's perspective, the execution of a program is regarded as a sequence of significant performance events. As the events are triggered during execution, the probes engage the TAU performance measurement infrastructure to obtain the performance data. Logically, instrumentation is separated from measurement in TAU. The measurement options will determine what performance data is recorded and the type of measurement made for the events, whether profile or trace. On the other hand, instrumentation is focused primarily on event creation and code insertion, how the events are created, where instrumentation is generated, and how code gets placed in the program. [1]

[1] The TAU User's Manual [46] gives full details on how to use the TAU instrumentation tools. Due to space limitations, we concentrate here instead on the instrumentation approach.

3.1 Event Interface

Instrumentation is accomplished by inserting probes in the source code of an application. The probes are generated from the TAU *event interface*, which allows events to be defined, their visibility controlled, and their runtime data structures to be created. Each event has type (atomic, interval), is part of an *event group*, and has a unique *event name* represented as a character string. The event name is a powerful way of encoding event information. At runtime, TAU maps the event name to a efficient *event ID* which is used elsewhere in the event interface.

It is important to understand the reasoning behind event names and IDs. IDs are integers that get generated on event creation and act as handles for events during measurement. However, assigning a uniform ID for the same event across *node:context:thread* boundaries is problematic. Event names make it possible to resolve different IDs to the same event. In addition, it makes *dynamic event generation* possible. In general, a new event can be created at any time during execution as long as the event name is unique for the thread of execution. This allows for instance, runtime context information to be used in forming an event name (*context-based events*), or values of routine parameters to be used to distinguish call variants, (*parameter-based events*).

In addition to the atomic and interval event types we discussed above, TAU defines a *phase* event to identify program phases. This is a type of "logical event" in the sense that it associates logical program aspects with an event, and distinguishes it through separate interfaces routines to the measurement system. They are equivalent to interval events, but have different measurement results. We discuss phases further below.

TAU also defines a *sample* event which is associated with an interrupt-based, measurement sampling procedure. A sample event acts like an atomic event, but is set up through a separate interface which connects it to an interrupt handler. Once enabled, an interrupt handler is invoked when the corresponding interrupts occur during program execution. Control of interrupt period and selection of system properties to track them are provided in the sample event interface.

The purpose of the event control in TAU is to enable and disable a group of events at a coarse level. This allow the focus of instrumentation to be refined at runtime. All groups can be disabled and any set of groups can be selectively enabled. Similarly, all event groups can be enabled initially and then selectively disabled. It is also possible to individually enable and disable events. TAU uses this support internally to throttle high overhead events during measurement.

3.2 Instrumentation Mechanisms

Instrumentation can be introduced in a program at several levels of the program transformation process. In fact, it is important to realize that events and event information can be between levels and a complete performance view may require contri-

bution across levels [37]. For these reasons, TAU supports several instrumentation mechanisms based on the code type and transformation level: source (manual, preprocessor, library interposition), binary/dynamic, interpreter, component, and virtual machine. There are multiple factors that affect the choice of what level to instrument, including accessibility, flexibility, portability, concern for intrusion, and functionality. It is not a question of what level is 'correct' because there are tradeoffs for each and different events are visible at different levels. The goal in the TAU performance system is to provide support for several mechanisms that might together be useful.

3.2.1 Source Instrumentation

Instrumenting at the source level is the most portable instrumentation approach. Source-level instrumentation can be placed at any point in the program and it allows a direct association between language-and program-level semantics and performance measurements. Using cross-language bindings, TAU implements the event interface in C, C++, Fortran, Java, and Python languages, and provides a higher-level specification in SIDL [20, 40] for cross-language portability and deployment in component-based programming environments[3].

Programmers can use the event API to manually annotate the source code of their program. For certain application projects, this may be the preferred way to control precisely where instrumentation is placed. Of course, manual instrumentation can be tedious and error prone. To address these issues, we have developed a powerful automatic source instrumentation tool, *tau_instrumentor*, for C, C++, and Fortran, based on the program database toolkit (PDT) [21]. The TAU source instrumentor tool can place probes at routine and class method entry/exit, on basic block boundaries, in outer loops, and as part of component proxies. PDT's robust parsing and source analysis capabilities enable the TAU instrumentor to work with very large and complex source files and inserts probes at all possible points.

In contrast to manual instrumentation, automatic instrumentation needs direction on what performance events of interest should be instrumented for in a particular performance experiment. TAU provides support for *selective instrumentation* in all automatic instrumentation schemes. An *event specification* file defines which of the possible performance events to instrument by grouping the event names in include and exclude lists. Regular expressions can be used in event name specifiers and file names can be given to restrict instrumentation focus. Selective instrumentation in TAU has proven invaluable as a means to customize performance experiments and to easily "select out" unwanted performance events, such as high frequency, small routines that may generate excessive measurement overhead.

Automatic source instrumentation is, in essence, a preprocessor style of instrumentation. There are other preprocessing instrumentation features implemented in TAU. For instance, we can instrument C *malloc*/*free* calls to use TAU's memory allocation/deallocation tracking package by redirecting the references to invoke TAU's corresponding memory wrapper calls with the added information about the

line number and the file. The atomic event interface is used in this case. I/O instrumentation is also implemented in this manner.

Library wrapping is a form of source instrumentation whereby the original library routines are replaced by instrumented versions which in turn call the original routines. The problem is how to avoid modifying the library calling interface. Some libraries provide support for *interposition*, where an alternative name-shifted interface to the native library is provided and weak bindings are used for application code linking. Here, library routines can be accessed with both its name shifted interface and the native interface. The advantage of this approach is that library-level instrumentation can be implemented by defining a wrapper interposition library layer that inserts instrumentation calls before and after calls to the native routines. It is also possible through interposition to access arguments passed to the native library.

Like other tools, TAU uses MPI's support for interposition (*PMPI* [13]) for performance instrumentation purposes. A combination of atomic and interval events are used for MPI. The atomic events allow TAU to track the size of messages in certain routines, for instance, while the interval events capture performance during routine execution. TAU provides a performance instrumented PMPI library for both MPI-1 and MPI-2. In general, automatic library wrapping, with and without interposition, is possible with TAU's instrumentation tools.

Source instrumentation can also be provided in source-to-source translation tools. TAU uses the Opari tool [24] for instrumenting OpenMP applications. Opari rewrites OpenMP directives to introduce instrumentation based on the POMP/POMP-2 [24] event model. TAU implements a POMP-compatible interface that allows OpenMP (POMP) events to be instantiated and measured. In general, source-to-source translation systems can provide a powerful infrastructure for instrumentation support.

3.2.2 Binary / Dynamic Instrumentation

Source instrumentation is possible only if the source code is available. TAU leverages other technologies to implement instrumentation support at the binary code level. In particular, DyninstAPI [8] is a dynamic instrumentation package that allows a tool to insert code snippets into a running program using a portable C++ class library. For DyninstAPI to be useful with the TAU measurement system, calls to the event API must be correctly constructed in the code snippets. TAU can then instrument a program at runtime, or alternatively it can re-write the executable image with instrumentation included. The current set of events available to TAU with DyninstAPI are limited to routine entry/exit. It is possible to use selective instrumentation for routine events. DyninstAPI creates a function mapping table to aid in efficient performance measurement. Code snippets are then inserted at entry and exit transition points in each routine. Dynaprof [26] is another tool that uses DyninstAPI for instrumentation and TAU for event creation and measurement.

3.2.3 Interpreter-Based Instrumentation

Interpreted language environments present an interesting target for TAU integration. Often such environments support easy integration with native language modules. In this case, it is reasonable to attempt to recreate the source-based instrumentation in the interpreted language, calling through the native language support to the TAU measurement system. However, it is also true that interpreted language environment have built-in support for identifying events and monitoring runtime system actions.

TAU has been integrated with Python by leveraging the Python interpreter's debugging and profiling capabilities to instrument all entry and exit calls. By including a TAU package and passing the top level routine as a parameter to the package's run method, all Python routines invoked subsequently are instrumented automatically at runtime. A TAU interval event is created when a call is dispatched for the first time. At routine entry and exit points, TAU's Python API is invoked to start and stop the interval events. TAU's measurement library is loaded by the interpreter at runtime. Since shared objects are used in Python, instrumentation from multiple levels see the same runtime performance data.

Python is particularly interesting since it can be used to dynamically link and control multi-language executable modules. This raises the issue of how to instrument a program constructed from modules derived from different languages and composed at runtime. Because TAU supports multiple instrumentation mechanisms, it is possible to use a combination of them for these types of interpreter-based, dynamically-composed applications.

3.2.4 Instrumentation of Component Software

Component technology extends the benefits of scripting systems and object-oriented design to support reuse and interoperability of component software, regardless of language and location [42]. A *component* is a software object that implements certain functionality and has a well-defined interface that conforms to a component architecture defining rules for how components link and work together [3]. It consists of a collection of *ports*, where each port represents a set of functions that are publicly available. Ports implemented by a component are known as *provides* ports, and other ports that a component is uses are known as *uses* ports.

The Common Component Architecture (CCA) [9] is a component-based methodology for developing scientific simulation codes. The architecture consists of a framework which enables components (embodiments of numerical algorithms and physical models) to work together. Components are peers and derive no implementation from others. Components publish their interfaces and use interfaces published by others. Components publishing the same interface and with the same functionality (but perhaps implemented via a different algorithm or data structure) may be transparently substituted for each other in a code or a component assembly. Components are compiled into shared libraries and are loaded in, instantiated, and composed into a useful code at runtime.

How should component-based programs be instrumented for performance measurement? The challenge here is in supporting an instrumentation methodology that is consistent with component-based software engineering. The approach taken with TAU for CCA was to develop a TAU performance component that other components could use for performance measurement. The TAU instrumentation API is thus recreated as the performance component's interface, supporting event creation, event control, and performance query. There are two ways to instrument a component based application using TAU. The first requires calls to the performance component's measurement port to be added to the source code. This is useful for fine-grained measurements inside the component. The second approach interposes a proxy component in front of a component, thus intercepting the calls to its provides port. In this case, for each edge that represents a port in the component connection graph, we can interpose the proxy along that edge. A proxy component implements a port interface and has provides and uses ports. The provides port is connected to the caller's uses port and its uses port is connected to the callee's provides port.

To aid in the construction of proxies, it is important to note that we only need to construct only one proxy component for each type of port. Different components that implement a given port use the same proxy component. To automate the process of creating a proxy component, TAU's proxy generator uses PDT to parse the source code of a component that implements a given port. It infers the arguments and return types of a port and its interfaces and constructs the source code of a proxy component, which when compiled and instantiated in the framework allows us to measure the performance of a component without any changes to its source or object code. This provides a powerful capability to build performance-engineered scientific components that can provide computational quality of service [33] and allows us to build *intelligent*, performance-aware components.

3.3 Instrumentation Utilities

To deliver the richness of instrumentation TAU provides for direct performance observation, it helps to have utilities to reduce the impact on users. Where this is most evident is in building applications with source instrumentation.

To simplify the integration of the source instrumentor and the MPI wrapper library in the build process, TAU provides a set of compiler wrappers: `tau_cc.sh`, `tau_cxx.sh` and `tau_f90.sh` that can be invoked instead of a regular compiler. For instance, in an application makefile, the variable `F90=mpxlf90` is modified to be `F90=tau_f90.sh`.

This tool invokes the compiler internally after extracting the names of source or object files and compilation parameters. During compilation, it invokes the parser from PDT, then the *tau_instrumentor* for inserting measurement probes into the source code, and compiles the instrumented version of the source to generate the desired object file. It can distinguish between object code creation and linking phases of compilation and during linking, it inserts the MPI wrapper library and the TAU

measurement library in the link command line. In this manner, a user can easily integrate TAU's portable performance instrumentation in the code generation process. Optional parameters can be passed to all four compilation phases.

Utilities are also helpful in reducing application reprogramming required just to get instrumentation enabled, such as when an external library must be instrumented without modifying its source. This may be necessary for libraries where the source is not available or for when the library is cumbersome to re-build. TAU's wrapper generator, `tau_wrap` may be used for such cases. This PDT-based tool will read header files of the original library and generate a new library wrapper header file with preprocessor `DEFINE` macros to change routine names to TAU routines names. A new wrapped library is created with these instrumented TAU routines, which then calls the original library. This is very similar to what is done for C `malloc`/`free` and I/O wrapping, except `tau_wrap` can be used for any library.

4 TAU Measurement

The measurement system is the heart and soul of TAU. It has evolved over time to a highly robust, scalable infrastructure portable to all HPC platforms. The instrumentation layer defines which events will be measured and the measurement system selects which performance data metrics to observe. Performance experiments are created by selecting the key events of interest and by configuring measurement modules together into a particular composition [11]. TAU provides portable timing support, integration with hardware performance counters, both parallel profiling and parallel tracing, runtime monitoring, and kernel-level measurement.

4.1 Measurement System Design

As shown in Fig. 1, the design of the measurement system is flexible and modular. It is responsible for creating and managing performance events, making measurements from available performance data sources, and recording profile and trace data for each *node:context:thread* in the execution. Compile-time and execution-time options govern measurement operation. In addition, runtime support is implemented to control TAU measurement function and focus during execution.

TAU implements a sophisticated runtime infrastructure for gaining both measurement efficiency and robustness. A core internal component is the runtime representation of the *event callstack* that captures the nesting relationship of interval performance events. The fact that the performance events are not required to be only routine entry/exit events makes the TAU event callstack a powerful measurement abstraction. In particular, the event callstack is key for managing execution context, allowing TAU to associate this context to the events being measured.

The TAU measurement system implements another novel performance observation feature called *performance mapping* [37]. The ability to associate low-level performance measurements with higher-level execution semantics is important to understanding parallel performance data with respect to the application's structure and dynamics. Performance mapping provides a mechanism whereby performance measurements, made for one instrumented event, can be associated with another (semantic) event at a different level of performance observation. TAU has implemented performance mapping as an integral part of its measurement system and uses it to implement sophisticated capabilities not found in other tools.

The core measurement support for parallel profiling maintains internal performance data structures for atomic and interval events, called the *profile table*. New events are created in the profile table by creating a new table entry, recording the event name, and linking in the storage allocated for the event performance data. What is important to understand is the profile table is generic, able to be used for all atomic and interval events, regardless of their complexity. Event type and context information can be recorded in event names and the TAU measurement system hashes and maps these names to determine when a new event has been created or already exists.

TAU's tracing infrastructure focuses on providing efficient, portable, and scalable trace record buffering and I/O. In recent years, it has also become important to interface the measurement infrastructure with sophisticated tracing libraries provided by VTF3 [36], OTF [19], and EPILOG [25]. In contrast to parallel profiling, the measurement system must be concerned with aspects of *node:context:thread* parallel interactions, especially with regard to timestamp synchronization.

4.2 Performance Data Sources

TAU provides access to various sources of performance data. Time is perhaps the most important and ubiquitous performance data metric, but it comes in various forms on different system platforms. TAU provides the user with a flexible choice of time sources based on system availability. At the same time, it abstracts the timer interface so as to insulate the rest of the measurement system from the nuances of different timer implementations. In a similar manner, TAU integrates alternative interfaces for access to hardware counters (PAPI [5] and PCL [4] are supported) and other system-accessible performance data sources. Through TAU configuration, all of the links to these packages are resolved.

Within the measurement system, TAU allows for multiple sources of performance data to be concurrently active, making it possible for both profiling and tracing to record multiple performance data. TAU also recognizes that some performance data may come directly from the parallel program. This is supported in two ways. First, the TAU API allows the user to specify a routine to serve as a counter source during performance measurement. Second, the TAU measurement system

supplies some internal events and counters that can be used to track program-related performance (e.g., tracking memory utilization and sizes of messages).

4.3 Parallel Profiles

Profiling characterizes the behavior of an application in terms of its aggregate performance metrics. Profiles are produced by calculating statistics for the selected measured performance data. Different statistics are kept for interval events and atomic events. For interval events, TAU computes exclusive and inclusive metrics for each event. The performance data here must be from monotonically increasing data sources (counter). Typically one source is measured (e.g., time), but the user may configure TAU with the `-MULTIPLECOUNTERS` configuration option and specify up to 25 metrics (by setting environment variables `COUNTER[1-25]`) to track during a single execution. For atomic events, the statistics measured include maxima, minima, mean, standard deviation, and the number of samples. When the program execution completes, a separate profile file is created for each *node:context:thread* instance. The profiling system is optimized to work with the target platform and the profiling operations are very efficient.

The TAU profiling system supports several profiling variants. The most basic and standard type of profiling is called *flat profiling*. If A is an interval event, flat profiles record the *exclusive* performance for A (i.e., performance while in A). Any time spent in events nested within A will be represented in A's profile as *inclusive* time, but it will not be differentiated with respect to the nested events. Flat profiles also keep information on the number of times A occurs and the number of times nested events occurs.

TAU can also generate parallel profiles that show performance with respect to event nesting (callstack) relationships. In general, *callpath profiling* determines the distribution of performance with respect to dynamic event nesting (calling paths) of an application. We speak of the *length* of a callpath as the number of events represented in the callpath (nesting chain). A callpath profile of length one is a flat profile. A callpath profile of length two is often referred to as a *callgraph profile*. A callpath of length k represents a sequence of $k-1$ nested events with an event, A, at the head of the callpath. The key concept to understand for callpath profiling is that a callpath itself is represented as a single performance event. TAU can generate a callpath profile of any length k (including $k = \infty$), producing a profile where every callpath of length $\leq k$ is represented.

TAU's callpath profiling will generate a profile for each callpath of a length designated by `TAU_CALLPATH_DEPTH`, not just those that include the topmost *root* event. For some performance evaluation studies, it is desired to see how the performance is distributed across program parts from a top-down, hierarchical perspective. Thus, a parallel profile that showed how performance data was distributed at different levels of an unfolding *event call tree* could help to better understand performance behavior. TAU's implementation of *calldepth profiling* does just that. It allows the

user to configure TAU with the `-DEPTHLIMIT` option and specify in the the environment variable `TAU_DEPTH_LIMIT` how far down the event call tree to observe performance. In this case, the profiles created show performance for each callpath in the rooted call tree pruned to the chosen depth.

While callpath profiling and calldepth profiling reveal the distribution of performance event based on nesting relationships, it is equally interesting to observe performance data relative to an execution *state*. The concept of a *phase* is common in scientific applications, its how developers think about the structural, logical, and numerical aspects of a computation, and therefore how performance can be interpreted. *Phase profiling* is an approach to profiling that measures performance relative to the phases of execution. TAU supports an interface to create phases (*phase events*) and to mark their entry and exit. Internally in the TAU measurement system, when a phase, *P*, is entered, all subsequent performance will be measured with respect to *P* until it exits. When phase profiles are recorded, a separate parallel profile is generated for each phase. Phases can be nested, in which case profiling follows normal scoping rules and is associated with the closest parent phase obtained by traversing up the callstack. When phase profiling is enabled, each thread of execution in an application has a default phase corresponding to the top level event. When phase profiling is not enabled, phases events acts just like interval events.

Recently, we have implemented support in TAU for recording of the current values of parallel profile measurements while the program is being executed. We call such a profile a *parallel profile snapshot*. The objective is to collect multiple parallel profile snapshots to generate a time-sequenced representation of the changing performance behavior of a program. In this manner, by analyzing a series of profile snapshot, temporal performance dynamics are revealed. Figure 2 shows a high-level view of the performance profile snapshot workflow.

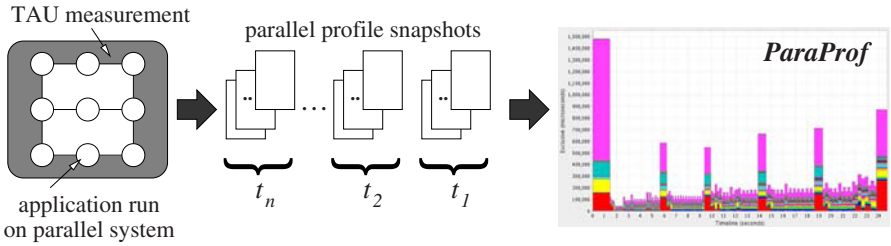


Fig. 2 Parallel profile snapshot process

4.4 Tracing

Parallel profiling aggregates performance metrics for events, but cannot highlight the time varying aspect of the parallel execution. TAU implements robust, portable,

and scalable parallel tracing support to log events in time-ordered tuples containing a time stamp, a location (e.g., node, thread), an identifier that specifies the type of event, event-specific information, and other performance-related data e.g., hardware counters). All performance events are available for tracing. With tracing enabled, every *node:context:thread* instance will generate a trace for instrumented events. TAU will write these traces in its modern trace format as well as in VTF3 [36], OTF [19], and EPILOG [25] formats. TAU writes performance traces for post-mortem analysis, but also supports an interface for online trace access. This includes mechanisms for online and hierarchical trace merging [7, 6].

4.5 Measurement Overhead

The performance events selected for observation depend mainly on those aspects of the execution that must be measured to satisfy the requirements for performance analysis. However, choice of performance events also depends on the scope and resolution of the performance measurement desired, as this impacts the accuracy of the measurement. The greater the degree of performance instrumentation in a program, the higher the likelihood that the performance measurements will alter the way the program behaves, an outcome termed *performance perturbation* [23]. In general, most performance tools, including TAU address the problem of performance perturbation indirectly by reducing the overhead of performance measurement.

We define *performance intrusion* as the amount of performance measurement overhead incurred during a performance experiment. We define *performance accuracy* as the degree to which our performance measures correctly represent "actual" performance. That is, accuracy is associated with error. If we are trying to measure the performance of small events, the error will be higher because of the measurement uncertainty that exists due to the relative size of the overhead versus the event. If we attempt to measure a lot of events, the performance intrusion may be high because of the accumulated measurement overhead, regardless of the measurement accuracy for that event.

Performance experiments should be concerned with both performance intrusion and performance accuracy, especially in regards to performance perturbation. TAU is a highly-engineered performance system and delivers excellent measurement efficiencies and low measurement overhead. However, it is easy to naively construct an experiment that will result in significant performance intrusion. TAU implements support to help the user manage the degree of performance instrumentation as a way to better control performance intrusion. The approach is to help the user identify performance events that have either poor measurement accuracy (i.e., they are small) or a high frequency of occurrence. Once these events are identified, the event selection mechanism described above can be used to reduce the instrumentation degree in the next experiment, thereby reducing performance intrusion in the next program run.

In addition, TAU implements two runtime techniques for profiling to address performance overhead. The first is *event throttling*. Here TAU regulates the active

performance events by watching to see if performance intrusion is excessive. Environment variables `TAU_THROTTLE_PERCALL` and `TAU_THROTTLE_NUMCALLS` can be set to disable events that exceed these thresholds at runtime. The second is *overhead compensation*. Here TAU estimate how much time is spent in various profiling operations. TAU will then attempt to compensate for these profiling overheads while these events are being measured. This is accomplished by subtracting the estimated amount of time dedicated to profiling when calculating time spent for an event. TAU can also compensate for metric besides time (e.g. floating-point operations). To enable measurement compensation the TAU measurement library must be configured with the `-COMPENSATE` option.

5 TAU Analysis

As the complexity of measuring parallel performance increase, the burden falls on analysis and visualization tools to interpret the performance information. If measurement is the heart and soul of the TAU performance system, the analysis tools bring TAU to life. As shown in Fig. 1, TAU includes sophisticated tools for parallel profile analysis and leverages existing trace analysis functionality available in robust external tools, including the Vampir [28] and Expert [49] tools. Let us focus on our work in parallel profile analysis and parallel performance data mining. The S3D [41] parallel application, a high-fidelity finite difference solver for compressible reacting flows, is used as an example.

5.1 Parallel Profile Management

The TAU performance measurement system is capable of producing parallel profiles for thousands of *node:context:thread* instances consisting of hundreds of events. Scalable analysis tools are required to handled this large amount of detailed performance information. Figure 3 shows TAU's parallel profile analysis environment. It consists of a framework for managing parallel profile data, *PerfDMF* [17] (expanded in the right figure), and the parallel profile analysis tools, *ParaProf* [2]. The complete environment is implemented in Java.

PerfDMF provides a common foundation for parsing, storing, and querying parallel profiles from multiple performance experiments. It supports the importing profile data from tools other than TAU through the use of embedded translators. These are built with PerfDMF's utilities and target a common, extensible parallel profile representation. Currently supported profile formats include gprof[15], TAU profiles[38], dynaprof[26], mpiP[47], HPMtoolkit (IBM)[10], and Perfsuite (psrun)[1]. Profile data can also exported as a common XML file.

The profile database component is the center of PerfDMF's persistent data storage. It builds on robust SQL relational database engines, including PostgreSQL[35],

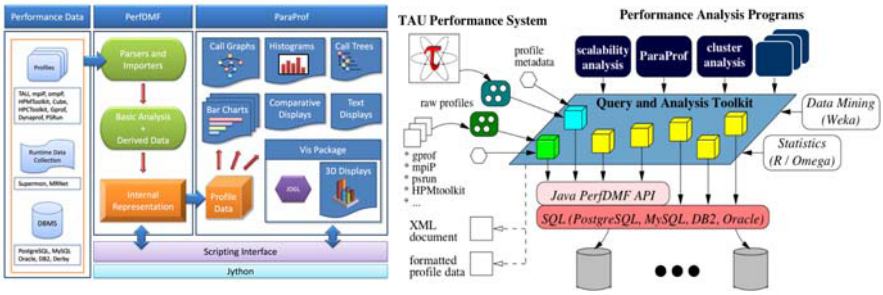


Fig. 3 TAU parallel profiling environment: ParaProf and PerfDMF

MySQL[27], Oracle[34], DB2[18] and Derby [14]. The database component must be able to handle both large-scale performance profiles, consisting of many events and threads of execution, as well as many profiles from multiple performance experiments.

To facilitate performance analysis development, the PerfDMF architecture includes a well-documented data management API to abstract query and analysis operation into a more programmatic, non-SQL, form. This layer is intended to complement the SQL interface, which is directly accessible by the analysis tools, with dynamic data management and higher-level query functions. It is anticipated that many analysis programs will utilize this API for implementation. Access to the SQL interface is provided using the Java Database Connectivity (JDBC) API.

5.2 Parallel Profile Analysis

The *ParaProf* parallel profile analysis tool included in TAU is capable of processing the richness of parallel profile information produced by the measurement system, both in terms of the profile types (flat, callpath, phase, snapshots) as well as scale. ParaProf provides the users with a highly graphical tool for viewing parallel profile data with respect to different viewing scopes and presentation methods. Profile data can be inputed directly from a PerfDMF database and multiple profiles can be analyzed simultaneously.

To get a brief sense of what ParaProf can produce, consider the S3D [41] parallel application as an example. Recently, we investigated the performance of S3D on a hybrid Cray system consisting of XT3 and XT4 processing nodes. Parallel performance profiles were produced when S3D ran on 6400 cores. ParaProf analyzed the full profile and generated the three-dimensional view shown in Fig. 4, left display. Due to the XT3 and XT4 memory performance differences, imbalances resulted in the S3D computation. These are clearly apparent in the ParaProf display. The right display in the figure is a ParaProf scatterplot showing clustering relationships for three significant events, color-code by processing core type.

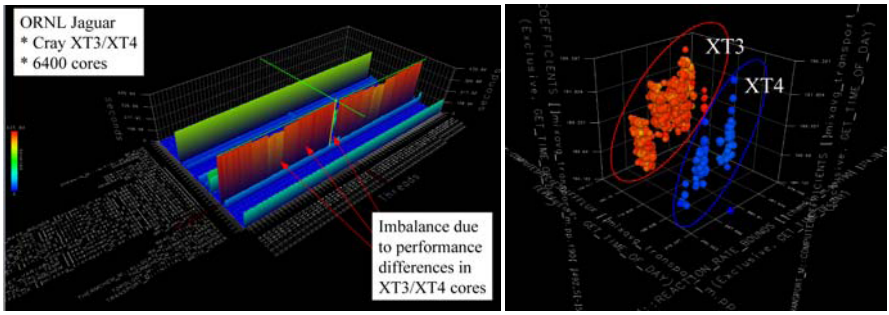


Fig. 4 ParaProf displays of S3D performance on hybrid Cray XT3/XT4 system

This example shows just a part of ParaProf's capabilities. ParaProf can shows parallel profile information in the form of bargraphs, callgraphs, scalable histograms, and cumulative plots. In addition, ParaProf can use these views to display profile snapshots (see Fig. 2, including with animation. ParaProf is also capable of integrating multiple performance profiles for the same performance experiment but using different performance metrics for each. Phase profiles are also fully supported in ParaProf. Users can navigate easily through the phase hierarchy and compare the performance of one phase with another. Figure 5 shows a phase profile display for the Uintah application. Here grid patches were used to identify different computational phases.

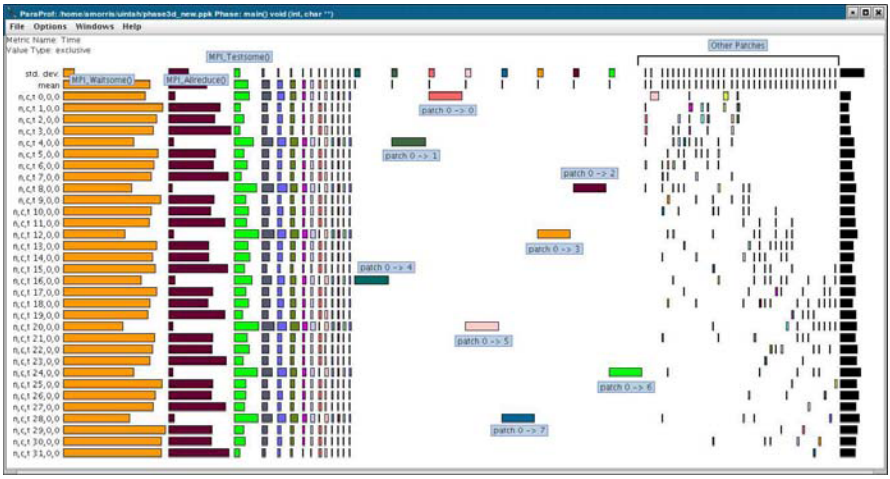


Fig. 5 Parallel profile phases for Uintah grid patches

ParaProf is able to extend its analysis functionality in two ways. First, it is capable of calculating derived statistics from performance metrics. A simple example of this is "floating point operations per second" derived from two metrics, "float-

ing point counts" and "time." Second, ParaProf analysis can be programmed with Python, using the Jython scripting interface.

5.3 Parallel Performance Data Mining

To provide more sophisticated performance analysis capabilities, we developed support for parallel performance data mining in TAU. *PerfExplorer* [16] is a framework for performance data mining motivated by our interest in automatic parallel performance analysis and by our concern for extensible and reusable performance tool technology. PerfExplorer is built on PerfDMF and targets large-scale performance analysis for single experiments on thousands of processors and for multiple experiments from parametric studies. PerfExplorer addresses the need to manage large-scale data complexity using techniques such as clustering and dimensionality reduction, and the need to perform automated discovery of relevant data relationships using comparative and correlation analysis techniques. Such data mining operations are engaged in the PerfExplorer framework via an open, flexible interface to statistical analysis and computational packages, including WEKA [48], the R system [43], and Octave [12].

A performance data mining framework should support both advanced analysis techniques as well as extensible *meta analysis* of performance results. Important components for productive performance analytics are:

1. *process control*: for scripting analysis processes
2. *persistence*: for recording results of intermediate analysis
3. *provenance*: mechanisms for retaining analysis results and history
4. *metadata*: for encoding experiment context
5. *reasoning/rules*: for capturing relationships between performance data

However, the framework must support application developers in the performance discovery process. The ability to engage in process programming, knowledge engineering (metadata and inference rules), and results management allows data mining environments to be created specific to the developer's concerns. PerfExplorer is being reengineered as shown in Fig. 6 to support these goals.

6 Conclusion and Future Work

The TAU Performance System® has undergone several incarnations in pursuit of its objectives: flexibility, portability, integration, interoperability, and scalability. The outcome is a robust technology suite that has significant coverage of the performance problem solving landscape for high-end computing. TAU follows a direct performance observation methodology. We feel this approach is best suited for parallel performance instrumentation, measurement, and analysis since it is based on

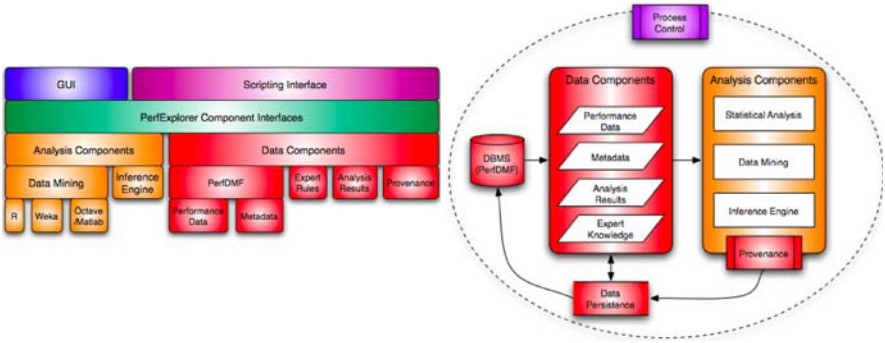


Fig. 6 TAU PerfExplorer data-mining architecture and component interaction

the observation of effects directly associated with the program's execution, allowing performance data to be interpreted in the context of the computation. However issues of instrumentation scope and measurement intrusion have to be addressed, but we have pursued these aggressively and enhanced the technology in several ways during TAU's lifetime.

TAU is still evolving. Although not reported here, we are adding support for performance monitoring to TAU which is built on scalable monitoring infrastructure from Supermon and MRNet [32, 30]. The goal here is to enable opportunities for dynamic performance analysis by allowing global performance information to be accessed at runtime. We are also extending our performance perspective to include observation of kernel operation and its effect on application performance [31]. This perspective will broaden to include parallel I/O and other sub-systems. Our vision here is to evolve TAU to do *whole performance evaluation* for petascale optimization. With the extreme scale and high integration in petascale platforms, it is difficult to see how reductionist approaches to performance evaluation will be able to support optimization and productivity objectives. Performance of petascale applications and systems should be evaluated in toto, to understand the effects of performance interactions and identify opportunities for optimization based on fully informed decisions. We intend to continue to evolve TAU to do so.

Acknowledgements This research is supported by the U.S. Department of Energy, Office of Science, under contracts DE-FG03-01ER25501 and DE-FG02-03ER25561. It is also supported by the U.S. National Science Foundation, Software Development for Cyberinfrastructure (SDCI), under award NSF-SDCI-0722072. The work is further supported by U.S. Department of Defense, High-Performance Computing Modernization Program (HPCMP), Programming Environment and Training (PET) activities through Mississippi State University under the terms of Agreement No. #GSO4TO1BFC0060. The opinions expressed herein are those of the author(s) and do not necessarily reflect the views of the DoD or Mississippi State University.

References

1. Ahn, D., Kufrin, R., Raghuraman, A., Seo, J.: Perfsuite. http://perfsuite.ncsa.uiuc.edu/
2. Bell, R., Malony, A., Shende, S.: A portable, extensible, and scalable tool for parallel performance profile analysis. In: Proc. EUROPAR 2003 Conference (EUROPAR03) (2003). URL http://www.cs.uoregon.edu/research/paracomp/papers/parco03/parco03.pdf
3. Bernholdt, D.E., Allan, B.A., Armstrong, R., Bertrand, F., Chiu, K., Dahlgren, T.L., Damevski, K., Elwasif, W.R., Epperly, T.G.W., Govindaraju, M., Katz, D.S., Kohl, J.A., Krishnan, M., Kumfert, G., Larson, J.W., Lefantzi, S., Lewis, M.J., Malony, A.D., McInnes, L., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Shende, S., Windus, T.L., Zhou, S.: A Component Architecture for High-Performance Scientific Computing. Intl. Journal of High-Performance Computing Applications **ACTS Collection Special Issue** (2005)
4. Berrendorf, R., Ziegler, H., Mohr, B.: PCL — The Performance Counter Library. http://www.fz-juelich.de/zam/PCL/
5. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications **14**(3), 189–204 (2000)
6. Brunst, H., Malony, A.D., Shende, S., Bell, R.: Online Remote Trace Analysis of Parallel Applications on High-Performance Clusters. In: Proceedings of the ISHPC Conference (LNCS 2858), pp. 440–449. Springer (2003)
7. Brunst, H., Nagel, W.E., Malony, A.D.: A Distributed Performance Analysis Architecture for Clusters. In: Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2003), pp. 73–83. IEEE Computer Society (2003)
8. Buck, B., Hollingsworth, J.: An API for Runtime Code Patching. Journal of High Performance Computing Applications **14**(4), 317–329 (2000)
9. CCA Forum: The Common Component Architecture Forum. http://www.cca-forum.org
10. DeRose, L.: The Hardware Performance Monitor Toolkit. In: Proceedings of the European Conference on Parallel Computing (EuroPar 2001, LNCS 2150), pp. 122–131. Springer (2001)
11. Dongarra, J., Malony, A.D., Moore, S., Mucci, P., Shende, S.: Performance Instrumentation and Measurement for Terascale Systems. In: Proceedings of the ICCS 2003 Conference (LNCS 2660), pp. 53–62 (2003)
12. Eaton, J.W.: Octave home page. URL http://www.octave.org/. Http://www.octave.org/
13. Forum, M.P.I.: MPI: A Message Passing Interface Standard. International Journal of Supercomputer Applications (Special Issue on MPI) **8**(3/4) (1994)
14. Foundation, T.A.S.: Apache derby. URL http://db.apache.org/derby/. Http://db.apache.org/derby/
15. Graham, S., Kessler, P., McKusick, M.: gprof: A Call Graph Execution Profiler. SIGPLAN '82 Symposium on Compiler Construction pp. 120–126 (1982)
16. Huck, K., Malony, A.: PerfExplorer: A performance data mining framework for large-scale parallel computing. In: Conference on High Performance Networking and Computing (SC'05) (2005)
17. Huck, K., Malony, A., Bell, R., Morris, A.: Design and Implementation of a Parallel Performance Data Management Framework. In: Proc. International Conference on Parallel Processing, ICPP-05 (2005)
18. IBM: IBM DB2 Information Management Software. http://www.ibm.com/software/data
19. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the Open Trace Format (OTF). In: Proceedings of the 6th International Conference on Computational Science, *Springer Lecture Notes in Computer Science*, vol. 3992, pp. 526–533. Reading, UK (2006)
20. Kohn, S., Kumfert, G., Painter, J., Ribbens, C.: Divorcing Language Dependencies from a Scientific Software Library. In: Proceedings of the 10th SIAM Conference on Parallel Processing (2001)

21. Lindlan, K.A., Cuny, J., Malony, A.D., Shende, S., Mohr, B., Rivenburgh, R., Rasmussen., C.: A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In: Proceedings of SC2000: High Performance Networking and Computing Conference (2000)
22. Malony, A., Shende, S.: Distributed and Parallel Systems: From Concepts to Applications, chap. Performance Technology for Complex Parallel and Distributed Systems, pp. 37–46. Kluwer, Norwell, MA (2000)
23. Malony, A.D.: Performance Observability. Ph.D. thesis, University of Illinois at Urbana-Champaign (1990)
24. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting. In: Proceedings of Third European Workshop on OpenMP (2001)
25. Mohr, B., Wolf, F.: KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications. In: Proceedings of the European Conference on Parallel Computing (EuroPar 2003, LNCS 2790), pp. 1301–1304. Springer (2003)
26. Mucci, P.: Dynaprof. http://www.cs.utk.edu/∼mucci/dynaprof
27. MySQL: MySQL: The World's Most Popular Open Source Database
28. Nagel, W., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer **12**(1), 69–80 (1996)
29. Nataraj, A., Malony, A.D., Shende, S., Morris, A.: Integrated parallel performance views. Cluster Computing **11**(1), 57–73 (2008). http://dx.doi.org/10.1007/s10586-007-0051-6
30. Nataraj, A., Morris, A., Malony, A.D., Arnold, D., Miller, B.: A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet. Under submission
31. Nataraj, A., Morris, A., Malony, A.D., Sottile, M., Beckman, P.: The Ghost in the Machine: Observing the Effects of Kernel Operation on Parallel Application Performance. In: ACM/IEEE SC2007. Reno, Nevada (2007)
32. Nataraj, A., Sottile, M., Morris, A., Malony, A.D., Shende, S.: TAUoverSupermon : Low-Overhead Online Parallel Performance Monitoring. In: Europar'07: European Conference on Parallel Processing (2007)
33. Norris, B., Ray, J., McInnes, L., Bernholdt, D., Elwasif, W., Malony, A., Shende, S.: Computational quality of service for scientific components. In: Proceedings of the International Symposium on Component-based Software Engineering (CBSE7). Springer (2004)
34. Oracle Corporation: Oracle. http://www.oracle.com
35. PostgreSQL: PostgreSQL: The World's Most Advanced Open Source Database. http://www.postgresql.org
36. Seidl, S.: VTF3 - A Fast Vampir Trace File Low-Level Management Library. Tech. Rep. ZHR-R-0304, Dresden University of Technology, Center for High-Performance Computing (2003)
37. Shende, S.: The Role of Instrumentation and Mapping in Performance Measurement. Ph.D. thesis, University of Oregon (2001)
38. Shende, S., Malony, A.D.: The TAU parallel performance system. The International Journal of High Performance Computing Applications **20**(2), 287–331 (2006). URL `http://www.cs.uoregon.edu/research/tau`
39. Shende, S., Malony, A.D., Cuny, J., Lindlan, K., Beckman, P., Karmesin, S.: Portable Profiling and Tracing for Parallel Scientific Applications using C++. In: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, SPDT'98, pp. 134–145 (1998)
40. Shende, S., Malony, A.D., Rasmussen, C., Sottile, M.: A Performance Interface for Component-Based Applications. In: Proceedings of International Workshop on Performance Modeling, Evaluation and Optimization, International Parallel and Distributed Processing Symposium (2003)
41. Subramanya, R., Reddy, R.: Sandia DNS code for 3D compressible flows - Final Report. Tech. Rep. PSC-Sandia-FR-3.0, Pittsburgh Supercomputing Center, PA (2000)
42. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (1997)

43. The R Foundation for Statistical Computing: R project for statistical computing (2007). URL http://www.r-project.org. Http://www.r-project.org
44. University of Oregon: TAU Portable Profiling. http://tau.uoregon.edu
45. University of Oregon: TAU Portal. http://tau.nic.uoregon.edu
46. University of Oregon: Tuning and Analysis Utilities User's Guide. http://www.cs.uoregon.edu/research/paracomp/tau
47. Vetter, J., Chambreau, C.: mpiP: Lightweight, Scalable MPI Profiling. http://www.llnl.gov/CASC/mpip/
48. Witten, ., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann (2005). URL http://www.cs.waikato.ac.nz/~ml/weka/
49. Wolf, F., Mohr, B., Dongarra, J., Moore, S.: Efficient Pattern Search in Large Traces through Successive Refinement. In: Proceedings of the European Conference on Parallel Computing (EuroPar 2004, LNCS 3149), pp. 47–54. Springer (2004)

Cray Performance Analysis Tools

Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon

Abstract The basic purpose of application performance tools, are to help the user identify whether or not their application is running efficiently on the computing resources available. However, the increasing system software and architecture complexity, as well as the scale of the current and future high end supercomputers, bring a new set of challenges to today's performance tools. In order to be able to achieve high performance on these peta-scale computing systems, users need a new infrastructure for performance analysis that can handle the challenges associated with heterogeneous architectures with multiple levels of parallelism, hundreds of thousands of computing elements, and novel programming paradigms. In this paper we present the Cray Performance Analysis Tools, which is set on an evolutionary path to address the application performance analysis challenges associated with these massive computing systems.

1 Introduction

The traditional way of conducting performance analysis and tuning for high performance computing has been an off-line approach with strong involvement from the user. A variety of performance measurement, analysis, and visualization tools have been created to help programmers tune and optimize their applications. These tools range from source code profilers [7, 5, 2, 9], to sophisticated tracers for analysis of communication [11, 8, 6, 13] and analysis of the memory subsystem [3], or a combination of the above [1]. These performance tools typically rely on a five-phase cycle, which consists of:

1. program *instrumentation* before execution
2. *measurement* of predefined specific events during execution

Cray Inc.
Mendota Heights, MN, USA

3. post-mortem and user-controlled *analysis* of the performance data
4. *presentation* of these data via textual and graphical tools
5. *optimization* of the program and its data structures under control of the user

The basic purpose of application performance tools, are to help the user identify whether or not their application is running efficiently on the computing resources available. To do this, most performance tools offer instrumentation, measurement and presentation components for use in the five-phase cycle, and a few tools have begun offering an analysis component to better assist users, where the most notable work in the area of performance analysis tools are Paradyn [10], from the University of Wisconsin, and KOJAK [12], from the Research Center Juelich.

The increasing system software and architecture complexity brings a new set of challenges to both today's users and performance tools. With systems designed to run hundreds of thousands of computing elements and support multiple levels of parallelism, the amount of performance data to collect, manage, and view has increased to the point where traditional performance measurement techniques aren't adequate. Current techniques can generate excessive amounts of information making it extremely difficult for users to correlate observations from data to understand performance behavior. In addition, the vast amounts of data generated for performance analysis degrades current tool response time and usability.

In order to be able to achieve high productivity with current and future high end computing systems, users need a new infrastructure for performance analysis that can handle the challenges associated with heterogeneous architectures with multiple levels of parallelism, hundreds of thousands of computing elements, and novel programming paradigms.

The Cray Performance Analysis Toolset, which includes instrumentation, measurement, analysis and presentation components for the performance analysis and code optimization cycle, is set on a revolutionary path to address the application performance analysis challenges associated with these massive computing systems.

This paper describes the Cray Performance Analysis Tools, including the special data analysis capability available that is designed to help the user identify potential performance bottlenecks that can benefit from optimization. We conclude the paper with a summary of next steps to be taken down that evolutionary path.

2 The Cray Performance Analysis Tools

The Cray Performance Analysis tools provide an integrated infrastructure for measurement and analysis of computation, communication, I/O, and memory utilization. The toolset allows developers to perform trace experiments on single-processor or multiple-processor executables at the binary level with function granularity. It supports all programming models.

The Cray Performance Analysis Tools consist of two components: the CrayPat Performance Collector and the Cray Apprentice^2 Performance Analyzer. CrayPat is the data capture tool, which is used to prepare user programs for performance

analysis experiments, to specify the kind of data to be captured during program execution, and to prepare plain text reports from the captured data or export the data for use with other programs. Cray Apprentice[2] is a post-processing data visualization tool that is used to further explore and study the resulting captured data. The focus of the Cray Performance Analysis Tools is on ease of use, flexibility, and intuitive user interfaces.

The Cray Performance Analysis Tools include both an external or asynchronous sampling mechanism, as well as an internal, or synchronous code instrumentation mechanism which inserts hooks within the application. The toolset allows data to be recorded either as a summation of events over time, or as a sequence of events over time. More details on when and how performance data is collected and recorded are discussed under the following tool component sections.

2.1 Program Instrumentation

When performing code instrumentation, CrayPat allows users to select the functions to be instrumented by group, by user function, or by name. Users do not need to modify the source code, the makefile, or even recompile the program to instrument at a function level. CrayPat also provides an API for fine-grain instrumentation, which requires recompilation after the insertion of the API calls. CrayPat uses binary rewrite techniques at the object level to create an instrumented application, which is generated with a single static re-link.

Information can be collected for functions, loops, regions, hardware events, data exchange via communication, synchronization, heap statistics, and I/O patterns. The instrumentation phase results in a standalone instrumented program that when executed, performs the collection and recording of performance data.

2.2 Data measurement

Performance data is captured during application execution by sampling, at time intervals or hardware counters overflow, or upon entry/return from traced functions, and is recorded in the form of a summarization of events over time (profile), or a sequence of events over time (trace). Each process collects its own performance data. Currently, internal buffers of per process memory are used to temporarily store local collected performance data. Once these buffers are full, they are flushed to a performance log file on a parallel file system.

The user can optionally control the behavior of the instrumented program during execution through a set of runtime environment variables that affect what and how the performance data is collected. Examples of this include the enabling of predefined hardware counter groups that track chosen sets of hardware events, the ability to choose the mechanism to use to sample the application, and the ability to modify

the number of data files that are written in parallel by the processes. By default, a runtime summarization of the data is provided, which includes aggregation of the data.

2.3 Analysis

Performance data analysis is offered for several areas. Examples include analysis and subsequent automation to alleviate cumbersome manual tasks associated with an experiment, and inter-node as well as intra-node data analysis.

The initial experiment, generally performed by a user is to determine where a program spends most of its time. The Cray Performance Analysis Tools have a feature called *"automatic profiling analysis"*, which performs a sampling experiment that creates a program profile, analyzes the profile to determine in which set of functions the program spent most of its time, and generates an input file that the user can use later for a more detailed experiment if needed. This is one example of data analysis that is performed prior to presentation, to help the user wade through potentially large amounts of data, and prepare them for more fine-grained experiments.

One of the key inter-node analyses performed on the raw data, is the detection of application load imbalance [4]. The components of the toolset can detect the percentage of resources available for parallelism that is "wasted". Examples of load imbalance include serial code segments that are executed by only one process, or global synchronization points where a slower process or processes delay the completion of a collective operation. Two imbalance metrics are calculated; the *"imbalance percentage"* and the *"imbalance time"*. The imbalance percentage is a range from 0 to 100, where a perfectly balanced code segment has an imbalance of 0%, and a serial code segment has an imbalance of 100%. Imbalance time, which relates to execution time, is provided to identify code regions that need optimization. It gives the user an upper-bound estimate for how much time could be saved in the overall program if the identified load imbalances were corrected.

Multi-core systems typically introduce a hierarchy with process communication. Some applications benefit from pairing processes on the same node which communicate often, while other applications benefit from combining computation and communication processes on a node to alleviate memory or network access bottlenecks. The toolset offers MPI rank placements suggestions based on analyzed memory traffic or sent message traffic to minimize communication overhead on multi-core systems.

Once the application developer has determined that there is a performance problem, the next step is to understand why that problem exists. The Cray Performance Analysis Tools support derived hardware counter metrics, which provide more intuitive performance characteristics for the user such as flops rate, % peak, cache utilization statistics, etc. Through these higher level metrics, the toolset helps identify the "why" to unexpected performance, so the application developer can more quickly identify the source of intra-node performance bottlenecks. Memory access

is one of the more readily addressable causes of intra-node performance bottlenecks. If data or instructions aren't in cache when the processor needs them, everything else stops while the system goes off and fetches the required information. In addition to derived metrics, a set of predefined hardware counter groups are available which collect meaningful sets of hardware counter events for the user. Choosing the group that provides cache utilization information tells the user how well their program is using the memory hierarchy.

Often HPC applications include loops that can benefit greatly from compiler optimization. Another analysis feature, known as Profile Guided Optimization, offered by the toolset for intra-node analysis, is the ability to work with a compiler to analyze hidden loop optimization potential. The tool reports loop count statistics, which gives the application developer a better idea of how often a sequence of events is repeated. Based on compiler feedback, it can also identify loops that have characteristics that prevent optimization, and offer suggestions for actions so the user can take advantage of vector instruction sets.

2.4 Presentation

The presentation component of the toolset currently includes both a text-based reporting utility: pat_report, as well as a GUI-based visualization tool: Cray Apprentice[2].

The pat_report utility available in the toolset performs two functions. It reads state and event data in the performance file created by the runtime library, and generates text reports according to the groups selected, presented in table format. Reports display such detail as hardware performance counters event values, call trees, and special processing for the function groups. One of the strengths of this utility is that it can be run several times against the same collected performance data to provide different combinations of data, so that the user can choose the subset from the collected data that best suits their needs. In addition to a predefined set of tables that can be generated, the utility supports options to control how the data will be aggregated and labeled.

Figure 1 shows one of the default tables generated by pat_report. This table shows the percentage of time and actual time spent in each function, along with the actual number of calls to each function and load imbalance metrics described above. It divides the MPI time into the the time for the MPI operations and the time that is spent in synchronization, which helps the user to identify load balance problems. For each group of functions (e.g., user, MPI, I/O), the table shows by default only functions that take at least 1% of the time. For example, this program spent 76.5% of its time in user functions, 16.6% of its time in MPI operations, 6.3% in synchronization of MPI collectives, and very little time in I/O and memory management. It shows a relatively high imbalance percentage of mpi_allreduce_, which could be an indication of load-balancing problem. However, the imbalance time of 0.27 seconds

```
Notes for table 1:

  Table option:
    -O profile
  Options implied by table option:
    -d ti%@1,ti,imb_ti,imb_ti%,tr -b gr,fu,pe=HIDE

This table shows only lines with Time% > 1.
    (To set thresholds to zero, specify:  -T)

  Percentages at each level are relative.
    (For absolute percentages, specify:  -s percent=a)


Table 1:  Profile by Function Group and Function

 Time % |      Time |Imb. Time |   Imb. | Calls |Group
        |           |          | Time % |       | Function
        |           |          |        |       |  PE='HIDE'

 100.0% | 7.193714 |       -- |     -- | 17604 |Total
|-----------------------------------------------------------
|  76.5% | 5.500078 |       -- |     -- |  4752 |USER
||----------------------------------------------------------
||  96.0% | 5.277791 | 0.171848 |   3.3% |    12 |sweep_
||   3.2% | 0.177352 | 0.005482 |   3.1% |    12 |source_
|===========================================================
|  16.6% | 1.197321 |       -- |     -- |  4603 |MPI
||----------------------------------------------------------
||  93.9% | 1.124227 | 0.277878 |  20.7% |  2280 |mpi_recv_
||   5.9% | 0.070481 | 0.014437 |  17.7% |  2280 |mpi_send_
|===========================================================
|   6.3% | 0.453091 |       -- |     -- |    39 |MPI_SYNC
||----------------------------------------------------------
||  61.1% | 0.277012 | 0.215608 |  45.7% |     4 |mpi_bcast_(sync)
||  38.7% | 0.175564 | 0.270049 |  63.2% |    32 |mpi_allreduce_(sync)
|===========================================================
|   0.5% | 0.037826 |       -- |     -- |  5992 |IO
||----------------------------------------------------------
||  99.2% | 0.037528 | 0.010681 |  23.1% |  5977 |ioctl
|===========================================================
|   0.1% | 0.005398 |       -- |     -- |  2218 |HEAP
||----------------------------------------------------------
||  52.8% | 0.002850 | 0.002219 |  45.7% |  1109 |malloc
||  47.1% | 0.002545 | 0.002388 |  50.5% |  1108 |free
|===========================================================
|===========================================================
```

Fig. 1 CrayPat profile by function group and function

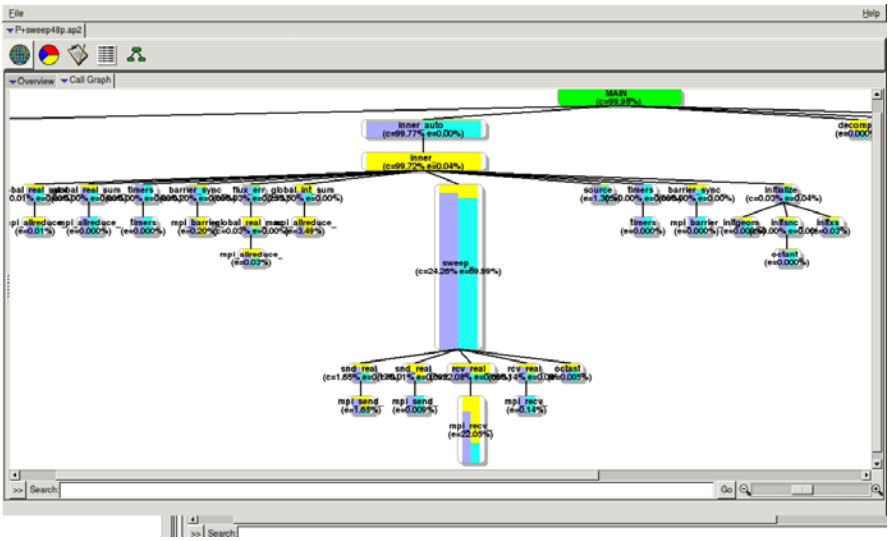


Fig. 2 Cray Apprentice^2 call graph view

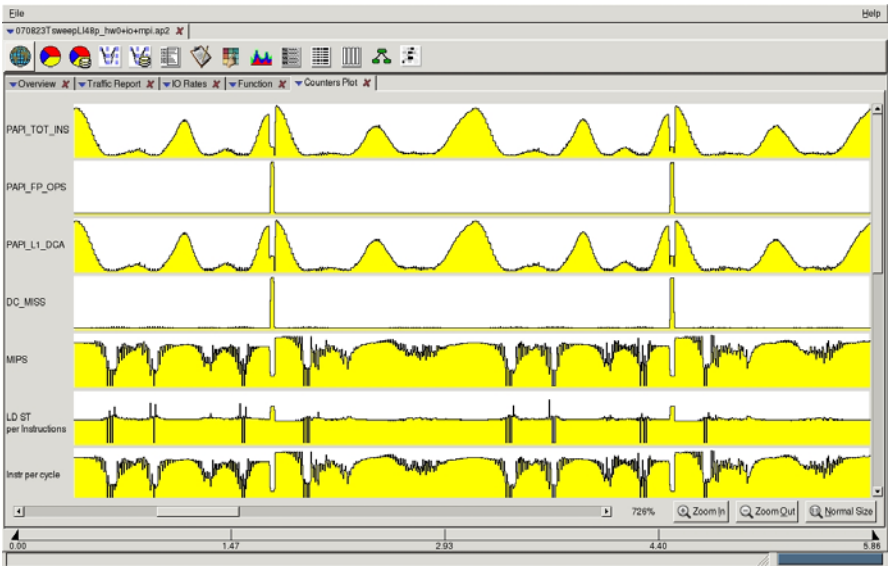


Fig. 3 Cray Apprentice^2 hardware counters view

gives an indication that it would not be worthwhile for the user to try to improve this load imbalance.

As its second function, pat_report combines the performance data collected at runtime with the information from the binary to generates a compressed perfor-

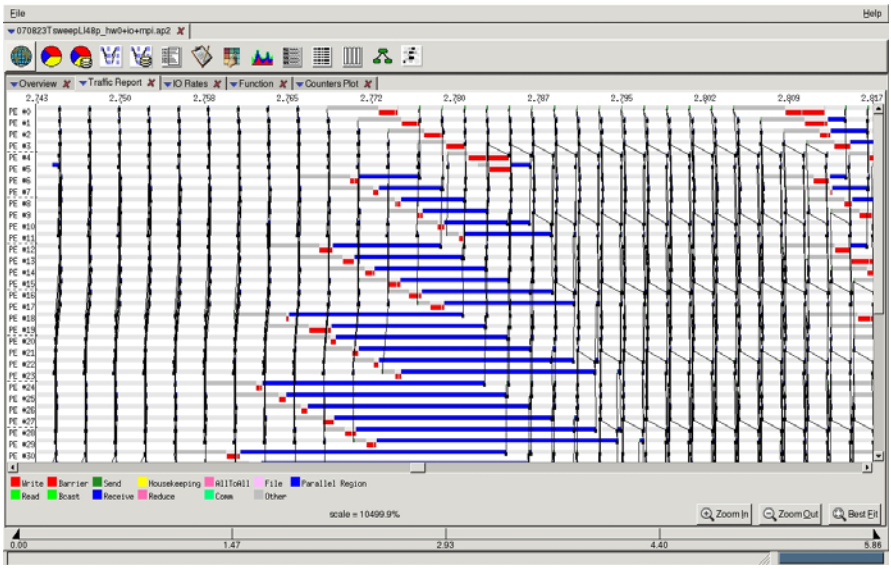


Fig. 4 Cray Apprentice^2 time line view

mance file that is used as input file by the Cray Apprentice^2 post-execution visualization tool. Cray Apprentice^2 is targeted to help identify performance bottlenecks, such as load imbalance, excessive serialization, excessive communication and network contention. It displays data captured by CrayPat during program execution. The GUI displays a variety of different data panels, depending on the type of performance experiment that was conducted with CrayPat. Cray Apprentice^2 provides call-graph-based profile information, as shown in Fig. 2, hardware counters displays, as shown in Fig. 3, and timeline-based trace visualization, as shown in Fig. 4. It supports traditional parallel processing and communication mechanisms, such as MPI, OpenMP, and SHMEM, as well as performance visualization for I/O, also displayed in Fig. 4.

3 Conclusions and Future Work

The Cray Performance Analysis Tools currently provide a robust infrastructure for application performance measurement and analysis. The goal of the next generation functionality is to offer a scalable solution to performance analysis, while further helping the user identify important and meaningful information from potentially massive data sets. The new functionality will extend Cray's existing performance measurement, analysis and visualization technology, as well as enhance user productivity by providing innovative techniques in areas such as data management and

further automated performance analysis to bring Cray optimization knowledge to a wider set of users.

Examples of enhancements include access to partial data specific to the type of analysis or presentation requested, data filters, a new data presentation hierarchy, and improved response time and usability of the presentation tools.

References

1. Bell, R., Malony, A.D., Shende, S.: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In: Proceedings of Euro-Par 2003, pp. 17–26 (2003)
2. DeRose, L.: The Hardware Performance Monitor Toolkit. In: Proceedings of Euro-Par 2001, pp. 122–131 (August 2001)
3. DeRose, L., Ekanadham, K., Hollingsworth, J.K., Sbaraglia, S.: SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In: Proceedings of SC2002. Baltimore, Maryland (2002)
4. DeRose, L., Homer, B., Johnson, D.: Detecting Application Load Imbalance on High End Massively Parallel Systems. In: Proceedings of Euro-Par 2007, pp. 151–159 (August 2001)
5. DeRose, L., Reed, D.: Svpablo: A Multi-Language Architecture-Independent Performance Analysis System. In: Proceedings of the International Conference on Parallel Processing, pp. 311–318 (1999)
6. European Center for Parallelism of Barcelona (CEPBA): Paraver - Parallel Program Visualization and Analysis Tool - Reference Manual (2000). Http://www.cepba.upc.es/paraver
7. Graham, S., Kessler, P., McKusick, M.: gprof: A Call Graph Execution Profiler. In: Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, pp. 120–126. Association for Computing Machinery, Boston, MA (1982)
8. Kim, S., Kuhn, B., Voss, M., Hoppe, H.C., Nagel, W.: VGV: Supporting Performance Analysis of Object-Oriented Mixed MPI/OpenMP Parallel Applications. In: Proceedings of the International Parallel and Distributed Processing Symposium (April 2002)
9. Mellor-Crummey, J., Fowler, R., Marin, G., Tallent, N.: HPCView: A tool for top-down analysis of node performance. The Journal of Supercomputing **23**(1), 81–104 (2002)
10. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tools. IEEE Computer **28**(11), 37–46 (1995)
11. Nagel, W., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: Vampir: Visualization and Analysis of MPI Resources. Supercomputer **12**, 69–80 (1996)
12. Wolf, F., Mohr, B.: Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing' **49**(10–11), 421–439 (2003)
13. Wu, C., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E., Gropp, W.: From trace generation to visualization: A performance framework for distributed parallel systems. In: Proceedings of Supercomputing 2000 (2000)

Index