13 Trends in Parallel Systems

In the earlier chapters of this book, we have studied the many architectural concepts which had been proposed and tried out until the early 1990s. In Chapter 12, we studied in some detail the basic issues related to instruction level parallelism (ILP), and the various techniques which have been developed to exploit ILP in the running program.

We shall now use that knowledge as a foundation to understand subsequent developments in computer architecture, in the light of the technological advances which have taken place over the last two decades. Of course this fairly brief chapter about the recent advances cannot possibly be *exhaustive*—but we do hope that it is *representative* enough to bring out the recent trends in computer architecture.

Over the last two decades, the hardware technologies that provide the building blocks of computer architecture have advanced almost beyond recognition. In Section 13.1, we shall take a brief look at these developments in technology, so as to understand the driving forces behind the recent developments in computer architecture. We feel that the recent innovations and advances in computer architecture cannot be studied in isolation of these technological factors.

In Section 13.2, we review in brief the types of parallelism which may be present in a program, and discuss the concept of *efficient* and *work-efficient* parallel algorithms. The concept of work-efficiency enables us to determine whether a given parallel algorithm has efficiency which is comparable to that of another known algorithm for the same problem. We also introduce the concept of *stream processing*, which can provide very high performance for certain specialized data-parallel applications.

In Section 13.3, we take a look at case studies of some recently introduced commercial processors and systems, which incorporate innovative designs based on the latest advances in technology and architectural concepts. In Section 13.4, we discuss current trends in parallel program development languages and techniques.

13.1 BRIEF OVERVIEW OF TECHNOLOGY

In electronics, VLSI, mass storage, and communication technologies, tremendous advances have taken place over the last two decades, which have shaped the resulting advances in processor and system architecture. In this section, we take a brief overview of these basic technological advances, so as to prepare the ground for case studies of some of the recently announced processors and systems. In sub-sections 13.1.1 through 13.1.4, respectively, we discuss semiconductor technology, display technology, storage technology, and interconnect and network technology.

13.1.1 Semiconductor Technology

Over the last several decades, steady advances in *very large scale integration* (VLSI) technology have led to a steady exponential-rate growth in the number of transistors which can be fabricated on a single chip. Present day technology allows well over a billion transistors to be fabricated on a single chip. Advances in VLSI technology have had a major impact on computer system architecture, giving rise to possibilities such as *multi-core chips* and *system-on-a-chip*.

The basic parameter which determines the size of a transistor on a chip is the minimum *line width* supported by the fabrication technology—i.e. the width of the smallest feature which can be fabricated on the chip.

With better and better processing technology, line widths producible using VLSI fabrication technology have been shrinking steadily. Sub-micron technologies became possible by the early 1990s, i.e. line widths of under a micron, which is 1000 nanometers (nm). Less than two decades later, we now have line widths of 65 nm, 45 nm, and even 32 nm, enabling the production of chips with over a billion transistors on them.

Gordon Moore was one of the founders of Intel Corporation, which is today the world leader in semiconductor technology and the largest manufacturer of semiconductor devices. Based on his intimate knowledge of VLSI design and fabrication technologies, Moore formulated *an empirical law* in the mid-1980s which states that: *The number of transistors which can be fabricated on a single chip doubles every two years.*

One way to understand the logic behind Moore's law is as follows:

(i) When a company embarks on developing 'the next generation' of chip technology, it typically aims for doubling of the device density on the chip. Since the area occupied by a device on the chip is proportional to the square of line width, the design target for the line width must be about $1/\sqrt{2}$ of the line width currently achieved. This approximate ratio explains the line widths of 90 nm, 65 nm, 45 nm etc. of current technologies.

(ii) The time period mentioned in Moore's law—two years—equals roughly the design and development cycle associated with the newer fabrication technology needed.

Faster clocks also become possible with improved technology; however, beyond a point, the power consumption of the chip rises disproportionately fast with clock speeds. Also, a faster processor clock requires an increased number of stages in processor pipelines. But there is a limit beyond which the number of such stages cannot be increased, because each additional pipeline stage introduces its own overhead.

In recent years, processor clock speeds have reached as much as 4 gigahertz, but it is seen that processor performance does not scale with clock speeds. One reason behind this is that the relative cost of a cache miss is greater at higher processor speeds.

In view of factors such as these, there has been a relative leveling off in processor clock speeds in recent years, while greater attention is given to how best to design the chip to utilize the enormous number of transistors on it. Apart from the exploitation of ILP discussed in Chapter 12, multi-core processors, systems-on-a-chip, stream processors, and larger two-level on-chip cache memories are other examples of resulting architectural developments.

An important consequence of high density chip designs and faster processor clocks is the following:

Off-chip interconnect delays play a major role in determining system performance. The approximate speed of an electronic signal over a wire in a computer system is 20 centimeters (cm) per nanosecond. If an off-chip connection has a length of 10 cm, for example, the associated delay is 0.5 ns, which is as much as half of a clock cycle of a 1 gigahertz clock, or one full cycle of a 2 gigahertz clock.

Given that a large number of transistors can be fabricated on a chip, it follows that huge performance benefits can be derived by integrating system functions *on a chip*, even if it is not possible to continue to push clock speeds higher. Another outcome of these technological factors is that system performance is more easily enhanced by employing multiple processors, than by pushing a single processor to its technological performance limits.

In the case studies which we shall consider later in this chapter, we shall see how different manufacturers have designed innovative high performance systems, keeping in mind the basic constraints of the underlying technology. We shall also see that today the most powerful computer systems in the world—such as Cray XT and IBM Blue Gene[1]—are based on the concept of *massively parallel processing*.

Another important effect of modern VLSI technology on computer architecture ensues from the economics of chip design.

Design costs associated with modern high performance processors are very high, which means that larger production quantities are needed to justify these costs. Therefore computer system architects today are more likely to make use of commercially available processors which are in volume production—i.e. commodity processors—while relying on innovations in *system design* to deliver higher performance. In fact massively parallel systems have been developed precisely to exploit the enormous amount of aggregate processing power which can be provided through the use of a large number of high-performance commodity processors operating in parallel.

In the case studies presented in Section 13.3, we shall see that advances in VLSI technology—which have been touched upon very briefly here—have had a major impact not only on processor designs but also on overall system architecture.

Semiconductor Memories Dynamic random access memory (DRAM), which provides the bulk of main memory in computer systems today, is also subject to Moore's law, i.e. doubling of transistor count on a single chip every two years. This means that a single memory chip today can store hundreds of megabytes of memory, and computer systems today are provided with main memories which are three orders of magnitude larger than in the early 1990s.

However, over the years, memory speed increases have not kept up with processor speed increases. Processor speeds have been increasing at a rate of over 50% per year, whereas memory speeds have been increasing at a rate of less than 10% per year.

Typical processor clock periods in the early 1990s were of the order of 25 nanoseconds, and memory cycle times of the order of 200 nanoseconds. Today these two numbers would be of the order of 1 nanosecond and 50 nanoseconds, respectively, which shows that, *relative to processor speeds*, main memory speeds are slower today. In such a system—unless something is done about it—the processor would see fifty idle clock cycles for every memory access on a cache miss, which is clearly not acceptable.

[1] All the product numbers and names used in this chapter are registered trademarks of the respective corporations named.

This means that the cost of a cache miss, counted in terms of number of processor cycles lost, is greater today than it was in the early 1990s. To put in another way, cache miss rate would have greater impact on processor throughput today than it did earlier.

In terms of computer system performance, this means that designers today have to rely on more innovative *memory latency hiding techniques*. We have already seen that multi-level caches, out-of-order instruction execution, and hardware multi-threading are some of the latency hiding techniques available to the system designer.

Even with various latency-hiding techniques, the memory sub-system must be capable of storing and delivering data at the required rates. Double data rate (DDR) devices, wider data paths, interleaving, and integrated L3 cache are some of the techniques employed for this purpose. High performance systems also employ memories with error correcting codes (ECC) to protect against random, one-off errors.

13.1.2 Display Technology

Graphics display technology has made huge strides since the mid-1990s—when LCD displays were virtually unknown, and high resolution CRT displays were only available on expensive workstations. In terms of each of the following performance features, graphics displays have made huge advances over the last couple of decades:

- pixel density,
- range of colors,
- contrast,
- refresh rate, and
- viewing angle (applicable to LCD displays).

With the help of specialized graphics controllers and high data rate interconnects, modern systems support animated graphics of amazing quality.

These developments have opened up the vast and entirely new area of multimedia applications, including animated graphics and sophisticated gaming—applications which were not possible a couple of decades earlier. Sophisticated image processing is also now made possible by utilizing the same processing and display capabilities.

At the same time, graphical interfaces have changed the ways in which users interact with the application programs. Compared to the earlier days of DOS and UNIX command lines, user interaction with the computer has been transformed with the help of windows, pointing devices and imaginative graphics.

Graphics controllers implement functions of the *graphics rendering pipelines*, which require repeated computations on sequences of integer or floating point operands. These numerical operands represent the 2D or 3D image which is being displayed, and the numerical operations carried out on them represent common graphics operations such as projection, clipping, scaling, rotation, and so on.

Computer graphics and image processing are highly specialized subjects, and it is not possible to delve into these subjects at this stage. But the processing and data transfer requirements of dynamic graphics do have a bearing on computer architecture, as the following example will illustrate.

Example 13.1 Graphics display: processing requirements

Consider a display of 1000 × 1200 pixels, with 24 bits of color information per pixel, i.e. 8 bits for each of the three primary colors; assume that the display has to be refreshed 60 times per second for animated graphics.

Then the aggregate data transfer requirements to the display can be calculated as:

$$1000 \times 1200 \times 24 \times 60 \text{ bits/second} = 216 \text{ MB/s.}$$

In a graphics system, all the subsystems—main memory, processor, graphics processor, as well as all the data paths —must support the required data rates.

For each pixel, display intensities of the three colors must be calculated. Let us assume this requires a hundred arithmetic operations per second on average, which may be integer or floating point operations, depending on system design. This figure is used here for an order-of-magnitude calculation; a more precise calculation requires details of the rendering process, and there would be potential parallelism in these operations.

Then the graphics processing power needed in this system is of the order of:

$$1000 \times 1200 \times 100 \text{ operations/second} = 120 \text{ million operations per second.}$$

Note that this processing power must in general be provided using the appropriate hardware technology, e.g. a pipelined graphics processing unit (GPU).

In a graphics or image processing system, all graphics data points are put through essentially the same sequence of arithmetic operations. Because of this, graphics processing has given rise to variants of SIMD architecture.

One such variant is today provided even on PCs of modest cost in the form of *streaming SIMD extension* to the Intel x86 instruction set (SSE, see Section 13.3.5). As another example of the impact of graphics and image processing on system architecture, we shall study the concept of *stream processing* (see Section 13.2.4).

Requirements of graphics processing have a major impact on the rest of the system design also, in terms of the storage, processing, networking, and I/O capability required. Multimedia traffic forms a major component of all Internet traffic, while the design of sophisticated video game consoles must also take into account the aggregate graphics processing requirements.

13.1.3 Storage Technology

Since the early 1990s, mass storage technology has witnessed technology innovations resulting in steady advances in the following respects:

- greater storage densities,
- smaller form factors,
- lower power consumption, and
- reduced costs.

Another significant technology innovation has been the development of *flash memories*, which are non-volatile solid state mass-storage devices.

As in the case of semiconductor devices, magnetic disks have also benefited from steady improvements in materials, processing and manufacturing technology. In addition, there have been breakthroughs—such as the use of giant magneto-resistive (GMR) effect, which has helped shrink the size of the read/write heads. Today, magnetic disk drives are available in 3.5" form factor with capacity 1 TB (1 Terabyte, i.e. 10^6 MB).

In addition, storage systems such as *redundant array of independent disks* (RAID) have been developed to make available on computer systems huge amounts of online disk storage, wherein a large number of physical drives appear as one logical storage unit. In a RAID storage system, the drives built into the array can provide the advantages of faster data access, error recovery, and/or fault tolerance.

To the operating system, a RAID storage unit appears exactly as one logical disk. The multiple physical disks in a RAID storage system may provide a combination of:

(i) *Data striping*—i.e. the data to be stored is distributed across multiple disks, so that it can be read or written in parallel across the disks, resulting in faster performance.

(ii) *Data mirroring*—all the data to be stored on one drive may be mirrored on another, so that operations continue uninterrupted even after a single disk failure.

(iii) *Parity*—for every $m \geq 2$ physical data disks, an extra physical disk may be used to store parity information calculated for the m data disks; in case of any parity violation detected, the system is in a position to provide error recovery.

These strategies can be combined. For example, one can have a redundant pair of striped disks, or a striped pair of redundant disks. Standards have been developed defining the various RAID configurations which are used to meet specific system objectives.

RAID features can be implemented in hardware, in which case the operating system views a RAID system just as it views any other disk drive. Alternatively, RAID features can also be implemented in software, in which case they make up the lower layer of the disk space management software.

However, even in the midst of rapid technological advances, one fact has remained unchanged over the years: Applications of computer systems invariably grow to stress and stretch the limits of available technology. We have already seen this to be true in the case of semiconductor and graphics technologies.

In the case of magnetic disk drives, the story is no different. Over the years, the data storage requirements of applications have grown exponentially. A large component of this storage is today *cyberspace*—i.e. millions of gigabytes of information made available to users around the world through the world wide web. In actual fact this cyber-space resides on thousands of *server farms*, each of which contains a large number of disk drives; this data is made available on the web through *web servers*.

Magnetic disk storage has traditionally provided far higher storage densities than non-volatile semiconductor *EEPROM*[2] storage. However, in recent years, storage densities of semiconductor *flash memories*—a form of EEPROM—have increased significantly, leading to their increased use with compact and mobile devices, where they offer a better alternative to magnetic disks. Unlike the original EEPROM devices, flash memories provide access to stored data on a block-wise basis.

Being semiconductor devices, such memories also benefit from the steady technology improvements summarized in Moore's law. The availability of high-density non-volatile semiconductor memories means that so-called *solid state drives* are now available, which can be used in place of magnetic disk drives. These

[2] Electrically erasable programmable read-only memory.

drives offer the advantages of higher throughput, lower latency, lower energy consumption, robustness and durability. It is likely that, in the coming years, solid state drives may replace some of the rotating magnetic disks as secondary storage devices.

13.1.4 Interconnect and Network Technologies

Within a computer system, the processor-memory and inter-processor interconnects, as also the data paths to network and device controllers, must sustain the data traffic rates needed for a given aggregate system performance. Latencies associated with the data paths also play a role in determining achievable system performance.

Modern supercomputers, data centers and server farms rely on high performance and high availability computing infrastructure in which interconnects play the crucial role. An interconnect within a computer system may span a single chip, a circuit module (or board), a single rack consisting of many circuit boards, or multiple racks spanning a distance of a few meters or few tens of meters. Thus an interconnect may be a *network on a chip* (NoC), a *system area network* (SAN), or something intermediate. Beyond the range of a system area network, a local area network (LAN) or a *wide area network* (WAN) is needed to interconnect systems into larger systems.

Within a system, with a larger number of processors being connected, there has been a shift from performance-limiting *shared media* interconnects—e.g. shared processor-memory bus—towards packet-based *switched media* interconnects, which make use of point-to-point links and routers. Such systems support higher aggregate bandwidths, and protocols for them are specially designed with low overheads and latencies.

HyperTransport An example of a high performance interconnect which has been developed to meet such system requirements is HyperTransport (HT)[3]—a point-to-point interconnect technology which is packet-based, scalable, and has low latency. HT Technology Consortium, consisting of several major hardware vendors, published the first version of this standard in 2001, while the latest version 3.1 has been published in 2009.

A useful feature of HT is that the command/address/data path width can be selected by the system designer to be 2, 4, 8, 16 or 32 bits[4]. The latest version of HT supports a maximum clock speed of 3.2 GHz and aggregate data transfer rates of up to 51.2 GB/s. The HT link can be directly provided on the processor/core, without requiring a separate interface device. The packet-oriented data transfer protocol is designed for low overhead and provides fast I/O interrupt processing, error retries and virtual channel support.

For achieving the high switching speeds needed, HT relies on the underlying physical layer based on the Low Voltage Differential Signaling (LVDS) standard[5], which offers advantages of low power consumption, higher speed, and the immunity to noise and interference which characterizes differential signaling.

Basic circuit theory tells us that a capacitive load can respond instantaneously to a step change in current, but not to a step change in voltage. The effective load in system interconnects—within a chip, or between chips on a circuit board—is capacitive, and therefore a current-driven signaling scheme can support faster data rates.

[3] See *http://www.hypertransport.org*

[4] PCI Express also provides a similar design option. See below.

[5] See *LVDS Owner's Manual*, 4th edition, published by National Semiconductor, 2008.

Consider the basic circuit shown in Fig. 13.1, in which a digital signal—i.e. a step change in current—is being communicated by the driver to the receiver. Clearly, the direction of current through the pair of wires (known as *current loop*) depends on whether transistors A & D or B & C are turned on.

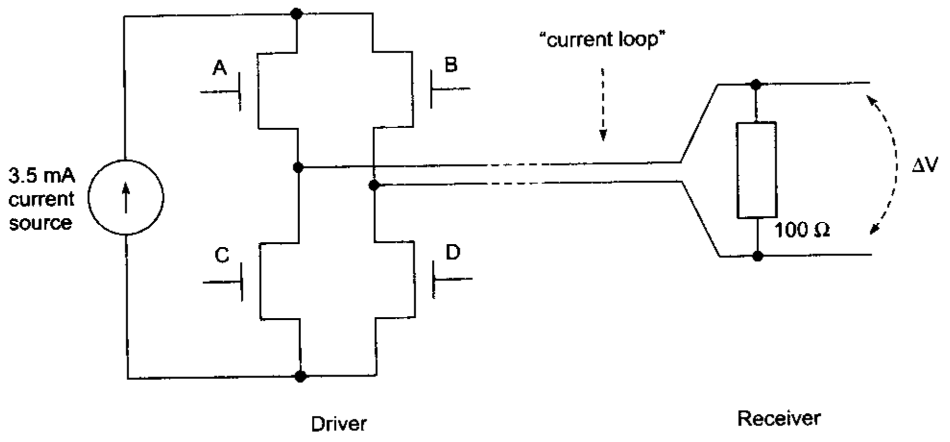


Fig. 13.1 Low voltage differential signaling (LVDS)

Across the 100 Ω terminating resistance, the 3.5 mA current source generates a voltage drop ΔV of 350 mV, given that any common mode voltage gets rejected in the differential arrangement. Data is recovered at the receiver from the sequence of changes in the polarity of this voltage.

PCI and PCI Express[6] *Peripheral Component Interconnect* (PCI) local bus standard was developed by Intel in the early 1990s for the relatively higher performance PCs then emerging—which were using, for example, Intel's own Pentium processors. The standard provides for device adaptors as IC chips on the motherboard, or as add-on cards in separate slots.

The original 32-bit version of PCI ran at 33.33 MHz clock speed, to deliver net data transfer rate of 133 MB/s. Later versions of PCI utilized × 2 and × 4 clock frequencies with proportionately faster data rates. The standard was extended for 3.3 volt operation, in addition to the original 5 volt definition, and a 64 bit version was also defined.

The PCI local bus can have a number of devices connected to it which can operate as *bus masters*. In case of multiple requests, a *bus arbiter* grants control to a single master; a pair of request/grant signals are provided for this purpose. The bus also includes address-cum-data lines and interrupt lines. Data transfer is carried out via *transactions*—in which an address phase is followed by data phase; read or write operations take place with respect to either memory address space or a separate I/O address space.

PCI has proved to be immensely successful, and has been introduced in several variants and form factors. It continues to be widely used in PCs, even after the enhanced and higher speed PCI Express standard was introduced in 2004.

PCI Express was introduced as a collaborative effort by Intel and other computer vendors in 2004. In spite of its similarity of name with PCI, it represents a radically different approach to system interconnects. PCI

[6] See *http://www.pcisig.com*

Express relies on serial, point-to-point links with message-based protocol implemented at the transaction layer. As a whole, a PCI Express based interconnect operates a set of independent and parallel point-to-point links, rather than the shared parallel bus of PCI. High speed graphics and storage devices which cannot use PCI are candidates for the use of PCI Express.

Over each pair of wires making up a single unidirectional link, current mode signaling is used to achieve data rate of 250 MB/sec in PCI Express v1.x, 500 MB/sec in v2.x and 1 GB/sec in v3.x. A pair of links make a bidirectional *lane*, and multiple (2, 4, 8, 12, 16 or 32) lanes can be configured together to achieve higher data rates, depending on the data transfer speed requirements of a device. Data carried over multiple lanes is *striped*, in the sense that in one transaction successive lanes carry successive bytes of data.

All data and control signals such as interrupts are sent as *messages* over the lane(s), rather than by using dedicated signal lines as in earlier systems (including PCI). The message based protocol uses CRC for error detection, and lower level ACK/NAK packets to signal message receipt or non-receipt (e.g. due to time-out); flow control for outstanding messages is provided at the transaction layer. Compared to PCI Express, HyperTransport (discussed above) uses a lighter, lower-latency message protocol.

As important as interconnect technology within a system is the local area and wide area networking technology which allows computer systems to communicate at high rates, even though they may be located half way around the world.

Today computer systems around the world are networked together in a way that could not even be imagined in the mid-1990s. Users have become accustomed to transferring huge amounts of data across the world at the press of a key, and most commercially important applications of computer systems rely on the availability of reliable, high bandwidth networks delivering services across much of the world.

When any type of data—numerical data, text, pictures, sound or video—is transferred between two or more computer systems, the quality of the underlying computer network is crucial in determining the overall system performance. Performance of a network link between computers is judged in terms of the bandwidth available, latency, and error rates. Of course errors can occasionally occur on links, and for this the network links provide for some form of error recovery. Performance of a network connecting two *end-to-end* systems can also be judged using essentially the same criteria.

Over the last decade, use of optical fiber technology has brought about a revolution in communication networks spanning the world. Achievable bandwidths have been rising, while costs have been coming down and the overall network reach has been increasing steadily. This has brought about a revolution in the type and range of applications which are being deployed and used routinely—applications which did not exist even a decade ago.

Gigabit Ethernet and Cluster Computing Ethernet, originally developed by Xerox Corporation, is the most widely deployed Local Area Network (LAN) around the world. IEEE Ethernet standard covers bottom Layers 1 and 2 of the seven-layer ISO protocol. The original standard, based on CSMA/CD technology, provided a speed of 10 Mbps; later Fast Ethernet with 100 Mbps speed became available. All along, Ethernet has proved to be an inexpensive, reliable, scalable and easily upgradeable LAN technology, leading to its huge adoption rate for local area networks and campus networks.

As processing power grows in the servers and user computers connected to a LAN, and as applications such as multimedia applications demand more bandwidth, the total traffic demands made on local area networks also increase.

In the late 1990s, IEEE defined the 1000 Mbps Ethernet standard—known as Gigabit Ethernet—first for fiber optic cables, and later over CAT-5 copper cables. Initially, it was envisaged to be used for network backbone, in the data center, and amongst the various traffic aggregation points in the network. But today it can also be provided on a user PC or workstation. The technology is defined by relevant parts of IEEE Standard 802.3.

The copper cable version of Gigabit Ethernet uses four pairs of standard Cat-5 cables, and explicit flow control amongst switches and adaptors, rather than CSMA/CD. Copper cables can be used over shorter distances, up to 100 m. Optical fiber cables can be used for longer distances, up to a few kilometers.

When multimedia applications share the same network as data intensive applications, the issue of network *Quality of Service* (QoS)[7] for each application becomes important; this can be seen from the following argument:

(i) A streaming video session needs timely delivery of data to avoid jitter, but it can tolerate occasional data errors.
(ii) On the other hand, for transfer of financial data, for example, absolute integrity is the prime concern, while some delay in delivery may be acceptable.

Thus different applications make different demands on the network for the required QoS; Gigabit Ethernet has been defined with the required support for the QoS concepts to be implemented over it. It should also be noted that, after Gigabit Ethernet, newer 10 Gigabit Ethernet technology has also now become available.

Availability of low-cost personal computers, high speed interconnects (such as InfiniBand and Gigabit Ethernet), and the use of the robust message-passing model to support concurrent processing have given rise to the popular and powerful *Cluster Computing* concept.

A cluster computer offers a low-cost alternative to supercomputers for obtaining higher processing power by interconnecting a large number of processing nodes. Technically, in terms of Flynn's original classification, Cluster Computing must be classified as *multiple instruction-stream, multiple data-stream* (MIMD) architecture, since each computer executes its own program. However, for a given application, if the same program is running over all the computers in the cluster, the processing is in *single program, multiple data-stream* (SPMD) mode.

The basic objective of employing a computing cluster may be *high performance*, *high availability* (i.e. ability to continue operating after a failure), or a combination of the two. High availability is made possible by providing redundancy in the system.

For example, a two-computer cluster, with both running the same database server, will provide higher availability than a single computer running the database server. On the other hand, for faster response to database queries, the database must be partitioned between the two interconnected computers running in parallel.

Partitioning and redundancy are two independent strategies—either or both may be adopted, depending on cost-benefit analysis. In the above example, if redundancy and partitioning are both needed, a total of at least four computers must be clustered.

[7] For a detailed discussion of this and related concepts, see for example *Computer Networks*, by Andrew Tanenbaum, fourth edition, Pearson Education.

For scientific and engineering applications, clusters of thousands of inexpensive computers have been built. However, programming them for a range of applications, and achieving peak theoretical performance, both remain challenges for the designer.

Beyond the cluster, Internet is now a world-wide phenomenon that is changing the world. Web-based applications, repositories of knowledge, and social networking have resulted in the creation of the vast *cyberspace*. With the use of the message-passing model, a network application runs correctly even though the respective clients and servers may be distributed around the world, although of course the response time seen by the user is dependent on the quality of the network links being used.

The message-passing model works equally well even amongst the multiple processors making up a single high performance computing system, being in this sense quite robust with respect to relative processor and communication speeds. Thus, with high performance interconnect and network technologies, newer models of parallel and distributed applications have evolved, enabling the enormous range of applications we see today.

Note 13.1

At this point, it is interesting to take a brief backward look at the kind of systems which were in use about fifteen years ago. Even this brief backward look makes clear the huge advances which have taken place in computer technology in the intervening period.

In the mid-1990s, the processors used in popular PCs were Intel 80386, 80486 and compatibles, running at clock speeds of at most a couple of hundred megahertz. Microsoft Windows 3.1 ran optionally on top of good old DOS.

The popular word processing software of those days was WordStar, and the commonly used spreadsheet software was Lotus 123, which had replaced VisiCalc. Microsoft Office was not yet available. While *UNIX* was in fairly common use, *LINUX* had not yet made its appearance.

LCD displays were not yet widely available, and there were no laptop PCs available as products. Spread of the Internet was very limited, and it was mostly used through UNIX-based programs such as *usenet* and *ftp*. The worldwide web was virtually unknown at the time, being in its stage of infancy, and therefore there were no web-based applications. Object-oriented programming with C++ was slowly gaining ground, while JAVA had not yet been introduced.

In the mid-1990s, there were no multimedia applications, no easy downloads of music or video files, and video games were of limited capability. The common local area network was based on 10 Mbps Ethernet, with Novell Netware[8] providing basic file storage and sharing services over the LAN.

From this brief summary, the amazingly rapid advances in computer technology over the last fifteen years become quite evident. In Sections 13.1.1 to 13.1.4, we have tried to identify some of the drivers of these advances.

13.2 FORMS OF PARALLELISM

We review in this section the main forms of parallelism which can be provided in a parallel processing system, relying on a basic division of parallelism between *structural parallelism*—

[8] An early product from Novell which was hugely successful in the market. For their more recent products, see *http://www.novell.com*.

i.e. algorithm level parallelism—and *instruction level parallelism*. This and some related points are discussed in Section 13.2.1.

The concepts of *work*, *work-efficiency* and *efficient parallel algorithm* are useful in parallel algorithm analysis and design, as is Brent's theorem. These ideas are discussed in Section 13.2.3. A simple parallel computation is presented earlier, in Section 13.2.2, to provide a basis for the discussion of Section 13.2.3.

Stream processing is a form of parallelism which emerges from a consideration of the type of processing involved in graphics, image processing, and signal processing. This form of parallel processing, which has some features in common with both SIMD and data flow models, has been discussed in Section 13.2.4.

13.2.1 Structural Parallelism versus Instruction Level Parallelism

In the previous few sections, we have taken a broad overview of the major advances which have taken place—over the last couple of decades—in processor, memory, storage, graphics, interconnect and network technologies. These advances have had a major impact not only on computer system architecture, but also on the kind of applications that are possible and are being demanded. Conversely, the growth in range of applications has also had an impact on how computer systems are designed and built.

In the study of high performance computer architecture, there is an important difference between *theoretical peak performance* of the system and the actual *performance achieved in practice*. This difference is often quite significant; for example, the performance achieved in practice, for solving a real-world problem on a highly parallel system, may be only 15% of the theoretical peak performance of the system.

This kind of a mismatch is not seen in other types of engineering products; for example, if a new model of a car is designed, and its actual performance in practice is only 25% of theoretical peak performance, the design will be judged a failure.

The basic reason for this type of performance mismatch in the case of highly parallel computer systems is the vast range of applications which are run on the systems. There are many possible application domains of such systems—such as scientific computations, engineering design and simulations, commercial and web applications, multimedia, games and virtual reality systems, signal processing, cryptography, and others.

Further, even within a given domain, there is a vast variety in the computational requirements of specific applications. The hard fact remains that, even for a single application in a given domain, it is a huge technical challenge to match its computational requirements to system hardware, and thereby achieve application performance approaching theoretical peak system performance.

In the earlier years of computer systems, the aim was to write programs which were provably correct—in the sense that they satisfied the specifications and were free from programming errors. For application programs running on a highly parallel system, we have an additional and important objective—that the programs make efficient use of all the computational resources available on the system.

In view of these facts, in utilizing high performance computer systems today, the technical challenge is to design applications with the most appropriate models of parallelism, so as to achieve the best possible performance.

The application is the final determinant of system architecture, in the sense that the architecture must necessarily serve the computing needs of the application. The application justifies the architecture. But, in fact, application requirements grow and evolve faster than system architecture, and therefore the challenge of matching growing application needs to evolving system architecture seems to be a never-ending one.

A few questions arise naturally in this context:

(1) Does the structure of the application have inherent, built-in parallelism in it?

For systems such as a web server or a transaction processing system, a large number of individual requests are processed almost independently of each other, and therefore parallelism can be exploited in the form of multi-threading. An independent thread can be created to process each service request or transaction. To support a large number of threads in parallel, the system must employ a proportionally larger number of processors, with multi-threading support within each processor.

For *data parallel* applications such as graphics rendering, or computation-intensive scientific and engineering applications, the SIMD or SPMD type model of parallelism may be more natural. Stream processing, discussed below, is also a variant of this type of parallelism. In some cases, parallelism is best exploited in the form of vector processing.

For any of such applications, the application designer and programmer(s) must explicitly design and develop the parallel program, using appropriate features provided in the programming language and the available function libraries. Such parallelism may be named *structural parallelism* in the application, which can only be exploited by the application designer and programmer(s) provided the system architecture has the necessary support for it.

Advances in parallel programming language design aim to enhance the power and expressivity of parallel programming—so as to facilitate the efficient realization of structural parallelism in an algorithm. The student is referred to Section 13.4.1, which describes the newly introduced parallel programming language Chapel.

(2) Can the compiler discover all the parallelism latent in the user's program?

The compiler *cannot* discover the structural parallelism in an application, of the type mentioned under (1) above. But, at the level of a single block of instructions, or across two or more blocks, the compiler may be able to discover potential parallelism and exploit it if the underlying processor architecture makes that possible. In addition, by techniques such as *vectorizing* and *loop unfolding*, the compiler may be able to bring out and exploit more of the latent parallelism in a program.

These points have been discussed in earlier chapters. In terms of instruction level parallelism, compiler-detection has its limitations, as we discussed in Chapter 12.

(3) Can the processor discover all the parallelism latent in the running program?

Clearly the processor cannot discover structural parallelism in a program, because such parallelism is not evident in the machine language version of the program. However, for exploiting parallelism in a block of instructions, or across two or more blocks, this would be the alternative to (2) to exploit the parallelism present in an instruction stream, as we have studied in Chapter 12.

As we have seen above, VLSI technology has provided the system designer with an abundance of hardware capabilities. Moore's law can be seen as one expression of the steady growth being achieved in VLSI capabilities. So now the obvious question facing the computer system architect is this:

For a given range of applications, and given the steady growth in VLSI capabilities, what should be the trade-off in processor and system design between supporting *structural parallelism* and *instruction level parallelism*?

The same question can also be posed in a slightly different way:

Suppose the designers of a new processor chip know that, with an improved VLSI process, they will have twice as many transistors in the next version of the chip. The designers must then resolve—at system level—the *trade-offs* between multiple cores, on-chip cache, functional units, pipeline stages, and aggressive exploitation of ILP. Clearly these system level trade-offs cannot be resolved without a clear picture of the target applications of the processor which is under design.

In recent years, there has been a shift in system design away from instruction level parallelism and towards support for structural parallelism. The basic driver behind the shift is simple: *to achieve maximum performance for a given system cost.* Development of multi-core architectures is a clear result of this shift, as is hardware multi-threading, and the provision of sophisticated, high-speed system interconnects.

Another important benefit of parallel architecture is the potential to provide *redundancy* to enhance the *fault tolerance* of a system. For a system which must provide 24 × 7 availability, an important benefit of having multiple processors, memories, and storage devices is that the system can continue to perform even in the presence of an occasional failure. As against *high performance*, this system characteristic is known as *high availability*.

As we have seen in earlier chapters, SIMD architecture and vector processing aim to exploit *data level parallelism* (DLP) in an application. Over the last two decades, processor designers have explored and developed every possible technique to exploit *instruction level parallelism* (ILP) in programs, to the point where scope for further progress in that direction seems to be limited.

In this scenario, the recent shift towards multi-core chips and hardware multithreading results in two types of important performance benefits:

(1) Multi-core chips and hardware multithreading can exploit a broader range of structural parallelism in applications. The processor cores in a multi-core chip operate in a shared memory mode. However, message-passing, which works independently of physical locations of processes or threads, also provides a natural software model to exploit the structural parallelism present in an application.

(2) A multi-core system with hardware multithreading also supports the natural parallelism which is always present between two or more *independent* programs running on a system. Even two or more operating systems can share a common hardware platform, in effect providing multiple *virtual computing environments* to users. Such *virtualization* makes it possible for the system to support more complex and composite workloads, resulting in better system utilization and return on investment.

We shall now take a look at a simple specific parallel computation, and then continue further with the discussion of parallel algorithms.

13.2.2 A Simple Parallel Computation

To visualize clearly the role of parallel processing in algorithms, we now consider a simple example. The computation here is a double-integration of a function of two variables over a rectangular region of the X-Y plane. The double integration is evaluated numerically using a simple parallel algorithm.

Example 13.2 Numerical integration over two variables

A continuous function $f(X,Y)$ of two variables X and Y defines a volume in the three-dimensional space created by the three axes X, Y and $Z = f(X,Y)$. This volume is determined by the integral:

$$\int_{Y\min}^{Y\max} \int_{X\min}^{X\max} f(X,Y)\,dX\,dY$$

where the appropriate limits on X and Y have been taken as Xmin, Xmax, Ymin and Ymax, respectively.

When such an integral is to be evaluated on a computer, the axes X and Y can be divided into intervals of length ΔX and ΔY, respectively, and the integral is replaced by the following summation:

$$\Sigma\Sigma f(X,Y)\,\Delta X\,\Delta Y$$

The function $f(X,Y)$ must be evaluated at an appropriate point, for example mid-point, within each area element of size $\Delta X \Delta Y$. Figure 13.2 illustrates graphically the double integration in question.

The number of intervals along X and Y axes is, respectively:

$$N_X = \frac{(X\max - X\min)}{\Delta X} \text{ and } N_Y = \frac{(Y\max - Y\min)}{\Delta Y}$$

Since the product $\Delta X \Delta Y$ is constant, it can be taken out of the summations. Function values $f(X,Y)$ need to be evaluated at various points within the grid which is formed on the X-Y plane by sets of orthogonal parallel lines drawn, respectively, at intervals of ΔY and ΔX. These grid lines define strips parallel to X and Y axes, with N_Y strips being parallel to X axis, and N_X strips being parallel to Y axis.

Thus, when represented as a computation, the volume integral reduces to a summation. Since the integral in question is a double integral over X and Y, the summation is also a double summation, with appropriate limits.

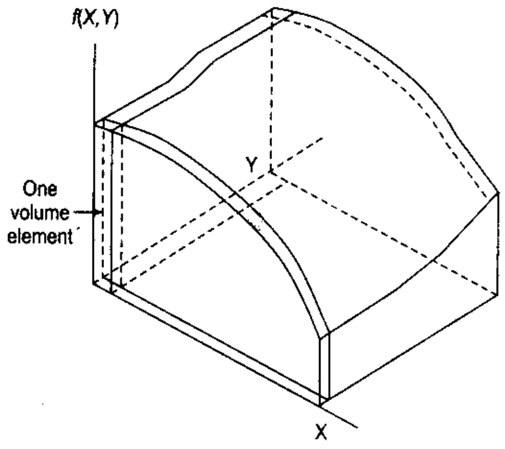


Fig. 13.2 Double integration of $f(X, Y)$ over rectangular region of X-Y plane

With a sequential algorithm, the summation requires $N_X N_Y$ evaluations of $f(X,Y)$ and the same number of addition steps, i.e. computation time is proportional to $N_X N_Y$.

One possible parallelized version of this algorithm is shown below[9]:

1. For each of the $N_X \times N_Y$ area elements, in parallel, calculate the value of the function $f(X,Y)$ at the mid-point of the area element.
2. For each of N_Y rows, in parallel, calculate the summation of $f(X,Y)$ at the N_X points along the row; denote this summation as the respective *row total*; this is the inner summation.
3. Calculate the sum of N_Y *row totals* found in step 2; this is the outer summation.
4. Multiply the sum of step 3 by $\Delta X \Delta Y$.

Note that $N_X \times N_Y$ processors are working in parallel in step 1. We shall discuss in Example 13.3 below the number of processors working in parallel in steps 2 and 3.

Note also that step 2 should not start until all processors have completed step 1, and similarly step 3 should not start until all the processors involved have completed step 2. As we have seen earlier, this type of synchronization between processors—or processes—is known as *barrier synchronization.*

We know that the addition of N numbers on a single processor takes N-1 addition steps. On multiple processors operating in parallel, we can perform the same addition of N numbers in a more time-efficient manner, as the following example illustrates.

Example 13.3 Addition using parallel processors

Let us consider the addition of $N = 8$ numbers on 4 processors. Assume that the numbers $a_0, a_1, \ldots, a_7$ are distributed on eight processors $p_0, p_1, \ldots, p_7$.

Step 1: Do in parallel: $a_0 + a_4 \rightarrow a_0$, $a_1 + a_5 \rightarrow a_1$, $a_2 + a_6 \rightarrow a_2$, $a_3 + a_7 \rightarrow a_3$

Note that here, when we say $a_0 + a_4 \rightarrow a_0$, what is meant is that the operand a_4 is made available from processor p_4 to processor p_0, using some mechanism of interprocessor communication. Operand a_0 is already present in processor p_0, and therefore the result of addition is also then available in processor p_0.

Step 2: Do in parallel: $a_0 + a_2 \rightarrow a_0$, $a_1 + a_3 \rightarrow a_1$

Step 3: $a_0 + a_1 \rightarrow a_0$

We see that four additions take place in parallel in step 1, two additions in step 2, and a single addition in step 3. Barrier synchronization is needed between steps.

Sum of the eight numbers is available in a_0 after *three* time steps, and the *degree of parallelism* is 4, since that is the maximum number of parallel operations we carried out, which was in step 1. Let us assume that, in general, $N = 2^k$ for some integer k, i.e. N is a power of 2. The student can easily verify that:

[9] The parallel algorithm is shown here only as an illustration. Better discussion of parallel algorithms can be found in relevant books; see, for example *Fundamentals of Sequential and Parallel Algorithms* by Kenneth Berman and Jerome Paul.

(i) In the above example, at the end of three time steps, variable a_0 in processor p_0 does indeed have the sum of the eight operands originally given to us.

(ii) In general, for $N = 2^k$ values to be added, the number of time steps required will be $k = \log_2 N$.

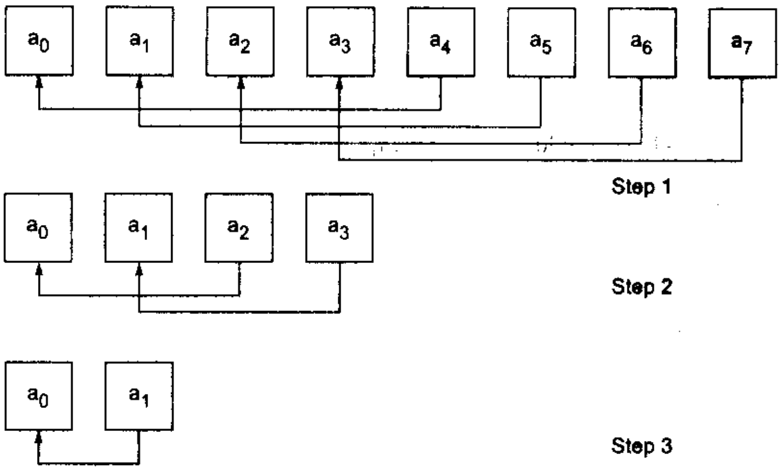


Fig. 13.3 Interprocessor communication in the three steps of the algorithm of Example 13.2, with $N = 8$

Figure 13.3 illustrates the pattern of communication between processors in the three steps of the above algorithm. Note also that this type of parallelism can be applied to any *associative operation* over N operands. For example, in the same way as addition was performed above, we could perform the *max* operation to find the largest of N operands, or multiplication to find their product.

To apply the basic concept of Example 13.3 to the double integration discussed in Example 13.2, let us assume for simplicity that we have a square grid over which the double integration is to be performed, i.e. $N_X = N_Y = N$. Then, for the steps of the parallel algorithm of Example 13.2, we can conclude that:

Function Evaluation: For the evaluation of $f(X,Y)$ at each grid element, we use N^2 processors, and the time taken is independent of N.

Row Totals: With $N/2$ processors used for each row, the N row totals can be calculated in parallel in $\log_2 N$ time steps.

Final Sum: The final sum is calculated using $N/2$ processors in $\log_2 N$ time steps.

Thus we see that, overall, with N^2 processors, the computation of double integration is performed in time $O(\log_2 N)$. In the next section, we shall discuss the issue of whether the parallel algorithm can be considered optimal with respect to the corresponding sequential algorithm for the same computation.

Figure 13.4 presents another depiction of the inter-processor communication of Example 13.3. We see that communication occurs in the pattern of a binary tree, with the addition—or any other *associative* operation—taking place at every internal node.

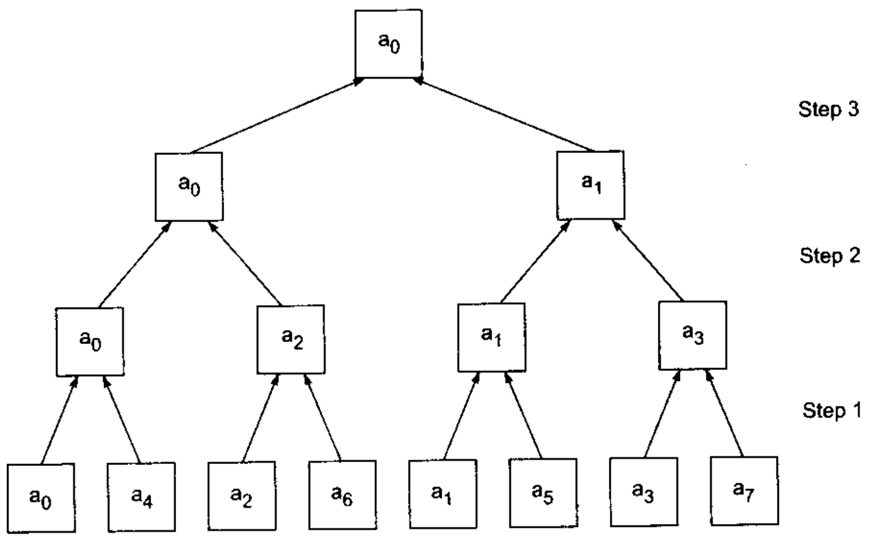


Fig. 13.4 Another depiction of the interprocessor communication in the three steps of the algorithm of Example 13.3, with $N = 8$

When such an operation is carried out on a multiprocessor system, it is known as a *reduce* or *reduction* operation. It involves addition in the above case, but the concept is more general, because any associative operation may be used as a basis for reduction.

13.2.3 Parallel Algorithms

As we know, the complexity of a sequential algorithm to solve a problem is defined in terms of the asymptotic running time of the algorithm on a problem instance of size n. This complexity is shown in 'big Oh' or 'order' notation, e.g. $O(t(n))$; this means that, for all values of $n > n_0$, the running time of the algorithm grows as $kt(n)$, for some constants n_0 and k.

When a number of processors work in parallel on a computation, we need to define the concept of the *work performed* by the algorithm. This necessarily depends on the number of processors used and the corresponding running time of the algorithm.

For a problem instance of size n, assume that an algorithm uses $p(n)$ processors in parallel and has running time in $O(t(n))$. Then the *work performed* by the algorithm on a problem instance of size n is defined as $w(n) = O(p(n)t(n))$[10].

In fact the actual number of processors used during the execution of a parallel algorithm often varies. In the summation of Example 13.3, we saw that the number of processors used decreases from $n/2$ to $n/4$, $n/8$, and so on. But we consider the maximum number of processors used at any step during the parallel execution, which is $n/2$ in that example.

[10] *Work performed* by the parallel algorithm can also be referred to as the *cost* of the algorithm; see the book cited above.

Now consider two different parallel algorithms, say I and II, for solving a given problem. In solving a problem instance of size n, let these two algorithms perform work $w_I(n) = O(p_I(n)t_I(n))$, and $w_{II}(n) = O(p_{II}(n)$ $t_{II}(n))$, respectively.

We say that algorithm I is *work-efficient* with respect to algorithm II if $w_I(n)$ is in $O(w_{II}(n))$, i.e. $w_I(n)$ is of the order of $w_{II}(n)$. Basically this means that, from the point of view of work performed, parallel algorithm I is comparable, within a constant multiplier, to parallel algorithm II.

A deterministic sequential algorithm is considered *efficient* is its running time $t(n)$ is a polynomial in n; bubble sort, for example, has running time in $O(n^2)$. For some problems, e.g. traveling salesperson or CNF satisfiability, no efficient i.e. polynomial running time algorithm is known—and it is conjectured, but not proven, that none exists.

In a similar way, we need to define the concept of an *efficient* parallel algorithm. Keeping in mind the parallel summation—or in general, reduction—of n elements, which we discussed in Example 13.3, we define an *efficient* parallel algorithm as follows:

A parallel algorithm is said to be *efficient* if, for solving a problem of size n, it satisfies the following two conditions:

(i) The number of processors $p(n)$ used is in $O(n^a)$, for some constant a, i.e. the number of processors required is polynomial in n, and

(ii) The running time of the algorithm $t(n)$ is in $O(\log^b n)$, for some constant b, i.e. the running time of the algorithm is *polylogarithmic* in n.

Note that the numerical integration algorithm of Examples 13.2 and 13.3 qualifies as an efficient parallel algorithm, with $a = 1$ and $b = 1$.

We can now go a step further and define an *optimal* parallel algorithm:

An *optimal* parallel algorithm is defined as one which is work-efficient with respect to the best possible sequential algorithm for solving the problem.

Consider finding the sum of n elements using $n/2$ processors in $\log n$ steps, as discussed in Example 13.3. Clearly the work done is $O(n\log n)$, and therefore this parallel computation is not work efficient with respect to the plain $O(n)$ sequential algorithm for summation. In fact this argument applies to any reduction operation carried out using an algorithm similar to that of Example 13.3.

Example 13.4

Consider Example 1.5, parallel multiplication of two $n \times n$ matrices. The first version of the algorithm uses n^3 processors and takes $O(\log n)$ time. Work done $p(n)\,t(n)$ is thus $O(n^3\log n)$, and therefore this algorithm is *not* work efficient with respect to the simple three-nested-loop sequential algorithm which runs in $O(n^3)$ time.

A modified version of the parallel algorithm is also presented in Example 1.5, which uses $n^3/\log n$ processors and runs in $O(\log n)$ time. Since the product $p(n)\,t(n)$ is now in $O(n^3)$, the modified algorithm is work efficient with respect to the $O(n^3)$ sequential algorithm.

Note that Strassen's sequential algorithm multiplies two $n \times n$ matrices in $O(n^{2.81})$ time, and in theory even more efficient algorithms exist for matrix multiplication. Therefore even the modified parallel algorithm of Example 1.5 cannot be considered optimal.

The student may recall that, for the second version of the algorithm in Example 1.5, the number of processors used is reduced by a factor of $\log n$, i.e. from n^3 to $n^3/\log n$. We may say that, in the second version of the algorithm, $n^3/\log n$ processors simulate the work of n^3 processors which are used in the first version.

In general, we can say that $q(n) < p(n)$ processors can simulate one time step of $p(n)$ parallel processors in $O(p(n)/q(n))$ time steps. Basically, each of the $q(n)$ processors can simulate the computation of $p(n)/q(n)$ processors, by executing instructions from that many instruction streams in a round-robin manner. For this, we must make the reasonable assumption that the 'context-switching' time during the simulation, from one instruction stream to the next, is constant, i.e. independent of n.

Using the argument of this type of simulation, we see that one time step of $p(n)$ processors translates into $O(p(n)/q(n))$ time steps of the $q(n) < p(n)$ processors. Thus the running time of the algorithm on the reduced $q(n)$ number of processors increases by a factor of $O(p(n)/q(n))$, giving us the theorem known as:

Brent's Theorem[11] For a given problem, suppose that there exists a parallel algorithm which solves a problem instance of size n using $p(n)$ processors in time $O(t(n))$. Further, suppose that we have $q(n) < p(n)$ processors available to solve the problem. Then the problem can be solved in time $O(p(n)t(n)/q(n))$.

In simple language, the simulation argument shows that, what we 'save' in terms of number of processors used, is spent on proportionately longer running time. Note that the two versions of the parallel algorithm, on $p(n)$ and $q(n)$ processors respectively, are work-efficient with respect to each other.

Amdahl's law (see Chapter 3) divides the computational requirements of an algorithm between the part which is parallelizable, and the rest which is not parallelizable—i.e. which must necessarily run as a sequential program. For the concept of work done, we have considered the largest number of processors used in parallel during the running of the algorithm. Therefore the point made in Amdahl's law has no direct bearing on Brent's theorem; both the theorems make valid statements about parallel algorithms.

To obtain the maximum possible time efficiency from a high performance processor or computer system, clearly the parallelism in the application must be discovered and then mapped onto the underlying hardware on which the application is to run.

In the previous section, we have seen a simple example of a parallel algorithm. Now, we can go a little further by posing questions such as the following:

- What is the nature of parallelism—*data parallelism* or *control parallelism*?
- Is the data parallelism in the algorithm amenable to stream processing, or is it more consistent with the SPMD mode of processing?
- In case of control parallelism, is it *fine grain* or *coarse grain* parallelism?

When we design and implement a parallel algorithm, clearly the program has explicit parallelism built into it. In Section 13.2.1, we have dubbed such parallelism as *structural parallelism*. For such programs,

[11] See *The parallel evaluation of general arithmetic expressions*, by Richard Brent, Journal of the ACM, vol. 21, no. 2, 1974. Of course Brent's theorem is not a recent development in computer architecture. However, because of its relevance in the design and performance of parallel algorithms, it has been included in this chapter.

the programming language—and/or the supporting library of functions—must allow program design using explicit parallel constructs. The source-level parallel program must then be mapped onto the hardware by the compiler and the library functions, and then supported by the runtime environment.

Clearly, the programming language, function libraries, runtime environment and the underlying hardware must all support the parallel constructs used. Some of the most demanding applications of high performance computer systems today are designed and implemented by intelligent exploitation of structural parallelism.

Applications designed to exploit hardware *multithreading*[12]—either *fine grain* or *coarse grain*—should also be considered as examples of *structural parallelism*. *Multiple instruction multiple data* (MIMD) parallelism, and the more restricted *single program multiple data* (SPMD) parallelism, both fall into this category, as does the recently introduced concept of *stream processing*.

We have discussed in the previous chapter (i) *instruction level parallelism* (ILP), exploited by the processor hardware while executing instructions, and (ii) compiler-detected parallelism, which is *implicit* in the application program. Clearly none of these forms of parallelism are involved at the stage of design of a parallel algorithm.

To clarify this point, we have shown in Fig. 13.5 the three typical stages in the process of writing, compiling and executing a parallel program. From this diagram, we see that:

(i) Structural parallelism enters into program design at the very first stage in program design and development, and it needs support from both the underlying stages. If we view program design in a *top down* manner, then this form of parallelism is introduced and exploited at the highest level of abstraction in program design.

(ii) Compiler-discovered parallelism is discovered in the second stage, and it needs support from the underlying hardware. This form of parallelism focuses on a block of instructions, or it may have scope spanning across two or more blocks.

(iii) Processor-discovered parallelism (ILP) is independent of the first two stages; it is discovered and exploited *on-the-fly* by the processor hardware; it relies on discovering independence between the multiple instructions of the program which occupy the fetch buffer and instruction pipeline at one time.

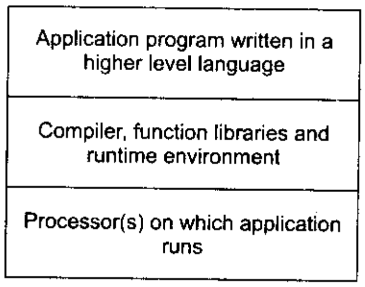

Application program written in a higher level language
Compiler, function libraries and runtime environment
Processor(s) on which application runs

Fig. 13.5 Stages in writing, compiling and running an application

[12] The word *thread* here may refer to a *process* as defined by the OS. To understand hardware support for multithreading, the distinction between *thread* and *process* is not crucial. The concept of hardware context applies to both—i.e. registers, PC, flags, etc. Here we assume that the OS takes care of the differences, if any, between a *thread* and a *process*.

Example 13.5

Presented below is a sequential version of the program of Example 13.2, which can easily be compiled and run on a conventional single-processor system. Note that Step 2 has a *nested loop* structure, which takes care of the $N_X \times N_Y$ grid in the *X-Y* plane.

```
1. Initialize SUM to zero
2. For I going from 1 to $N_X$
       For J going from 1 to $N_Y$
           Calculate the value of the function
           $f(X[I]+ \Delta X/2, Y[J]+ \Delta Y/2)$ and add it to SUM.
3. Multiply SUM by $\Delta X \Delta Y$.
```

Is it possible that a compiler could parallelize this program? The general aim is that, for a system with N_p processors, a parallelizing compiler will produce code which will carry out the above computation in time proportional to $N_X \times N_Y / N_p$; here N_p need not in general be related to N_X or N_Y, except that it is less than or equal to $N_X \times N_Y$.

If the compiler generated code satisfies this condition, then the parallelized version of the algorithm is work-efficient with respect to the sequential version.

In this particular program, since loop iterations are independent of each other, we may concede that this program can thus be parallelized.

In the general case, however, this is certainly a non-trivial task, and no compiler can extract the maximum degree of parallelism from an arbitrary sequential program. Chapter 10 of this book discusses several relevant techniques. On some systems, for example *openMP* (see Section 13.4.2), the programmer can pass a 'hint' to the compiler when a loop is to be parallelized, and the compiler can then do the needful.

Parallel programming languages—for example, the newly developed Chapel (see Section 13.4.1)—provide explicit parallel program structures. Therefore in such cases the job of the compiler does not involve *detection* of parallelism, but only efficient code generation for the target parallel hardware platform.

SIMD and MIMD forms of parallelism have been discussed quite extensively in the earlier chapters of this book. Over the last decade and a half, with newer technology being available and greater demands being made on systems, a new form of parallelism has been put to use on a wide scale in applications, which has been dubbed *single program multiple data* (SPMD).

Parallelism in this case can take the form of *one independent thread of execution per task, request or transaction to be processed*. Unlike in the traditional SIMD model, there is no lock-step synchronization here between the multiple threads, and there need not be *one-to-one* relationship between threads and processors. We may even assume here that the threads execute the same reentrant code. Therefore the SPMD model of parallelism has the advantages of simplicity of implementation and easy scalability, and the model has achieved wide-spread use for commercial and server-based applications.

Stream processing is another form of parallelism, proposed and developed in recent years, which has some characteristics of SIMD as well as data flow processing. This is a form of data parallelism which relies on high level of data locality and regularity in the processing of stream data, and can yield huge performance benefits, as we shall see in some more detail in Section 13.2.4.

Each of the forms of parallelism has its advantages and difficulties. With the background we have gained thus far, in Section 13.3 we shall look at a few case studies of recent developments in processor and system architecture, and in program development tools and techniques. Some of these case studies are in continuation of the systems studied in earlier chapters, while others are new entrants.

The important point bears repetition that the technological advances outlined briefly earlier in this chapter have had a very important bearing on the developments in computer hardware and software technologies. As the supporting technologies advance further, we shall no doubt see further innovations in computer architecture and technology as well.

13.2.4 Stream Processing

For animated 3D graphics, multimedia, image and signal processing applications, very high processing power is needed—and data is processed mostly in the form of *data streams*. Sometimes animated graphics includes the simulation of *game physics*, i.e. simulation of multiple objects in the scene behaving under modeled physical laws. Other applications where stream processing can be useful include 3G mobiles, set-top boxes, biological computations, cryptography, and database queries.

All data elements in a data stream go through the same processing stages. For example, the 3D graphical model of a car may be made up of hundreds of thousands of line elements or polygons, which must be processed through the so-called *rendering pipeline* to display the car on the system display screen. For animated graphics, a certain number of picture frames must be processed and displayed per second, and in general each frame must be processed through the same rendering pipeline.

These huge demands made on processing capability are for a single application—e.g. graphics processing, including game physics. To cater to these highly specialized needs, *graphics processing units* (GPUs) have been developed over the years, to relieve the main general-purpose processor(s) in the system of graphics processing load. The GPU operates in parallel with other processor or processors in the system.

With advances in VLSI technology, GPUs have also grown in processing power. Several research groups and commercial producers of GPUs have therefore sought to apply the vastly increased processing power of GPUs to more general computing. This has led to the emergence of *stream processing*, which combines high processing power, energy efficiency and programmability by exploiting the key properties of *data parallelism* and *data locality* which characterize data streams.

Stream processing can be seen as a new variant of SIMD, in which streams of data flow amongst *processing kernels*; in this sense, stream processing involves also some features of data flow processing. The *processing kernels* are basically software functions being executed on GPU processor cores. Multiple copies of a kernel execute in parallel on multiple cores—thus giving SIMD character to this form of processing.

The basic concept is illustrated in Fig. 13.6, with four kernels operating on one data stream. Multiple such sets of kernels will in general execute in parallel in a stream processor, to achieve proportionately higher parallel processing power. With sixteen such sets operating in parallel, for example, the total number of processor cores employed will be 64, with each of the four kernel functions executing in sixteen cores in parallel. Note that the number of data streams being processed in parallel will be sixteen.

A so-called *local register file* is provided with each core to maintain copies of working variables for the single execution *thread* (or *task*) running in each core. There is no multi-threading provided in each core, but it is possible to exploit ILP to some extent within each core. Data locality plays a key role in the design of a stream processing algorithm; stream processing is a form of structural parallelism, as defined in Section 13.2.1.

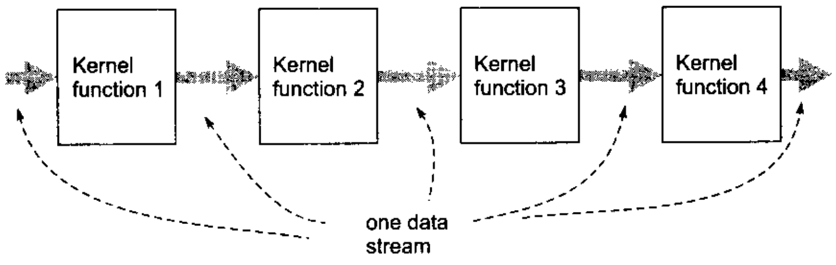


Fig. 13.6 Four processing kernels operating on a data stream

The properties of *data parallelism* and *data locality* govern the design of stream processors, since they permit efficient use of the bandwidth to memory—without the use of huge and expensive cache hierarchies. Recall that cache hierarchies are designed to support any random pattern of accesses to main memory, whereas memory accesses made for data streams are in a highly regular pattern.

Researchers at Stanford University designed the IMAGINE stream processor, which achieved tens of Gflops performance for certain graphics applications—with aggregate power dissipation less than 10 watts. MERRIMAC is the name of another research project at Stanford aimed at a larger computing platform using stream architecture and advanced interconnection networks. This research project had goals of achieving a high ratio of computation to communication, very high performance, compact size, high energy efficiency, reliability, simple system management, and scalability[13].

Nvidia Corporation[14] has long held a leading position in industry as a producer of graphics processing units. As GPUs grew in processing power, Nvidia developed more general-purpose processors based on their graphics expertise. They also defined a hardware/software platform named *Compute Unified Device Architecture* (CUDA) for general purpose program development using GPUs and standard programming languages. Nvidia named this concept *GPU Computing*—i.e. GPUs applied to general purpose computing.

From around 2006, Nvidia have developed several multi-core, multi-threaded *general-purpose GPUs* (also called GPGPUs), which were named GeForce, Quadro and Tesla. With substantial improvements, Nvidia have now announced their advanced Fermi architecture for GPU Computing.

The first Fermi based GPU from Nvidia has over 3.0 billion transistors and 512 cores. Each core executes a floating point or integer instruction per clock. The 512 cores are organized in 16 so-called *stream multiprocessors* (SMs) of 32 cores each. L2 cache is shared between the 16 SMs. The GPU chip provides six

[13] See *http://merrimac.stanford.edu* and *http://cva.stanford.edu*

[14] See *http://www.nvidia.com*

64-bit memory interfaces, for a total 384-bit memory interface, supporting up to a total of 6 GB of memory. A host interface connects the GPU to the CPU via PCI-Express, while the GigaThread unit on the GPU schedules *groups of threads* amongst the SMs.

A schematic diagram of the Fermi chip is shown in Fig. 13.7. Apart from the 32 cores, each SM is also provided with 16 *load/store units*, and four independent special function units (SFUs) to compute sine, cosine, reciprocal, and square root functions. The cores themselves are very basic, with one ALU and one FPU each.

Compared to earlier GPUs developed by Nvidia, Fermi offers improved memory access and double-precision floating point performance, ECC support, (limited) cache hierarchy, more shared memory amongst SMs, faster context switching, faster atomic operations and instruction scheduling, and the use of predication to reduce branch penalty. Threads are grouped into larger units—known as warps, blocks, grids—for the purpose of scheduling.

Most of the area in the Fermi chip is taken up by actual processing elements—i.e. FPUs, ALUs, and SFUs. This is unlike more conventional processors, in which huge cache memories occupy a greater proportion of chip area. This basic difference accounts for the higher processing performance and energy efficiency of stream processors.

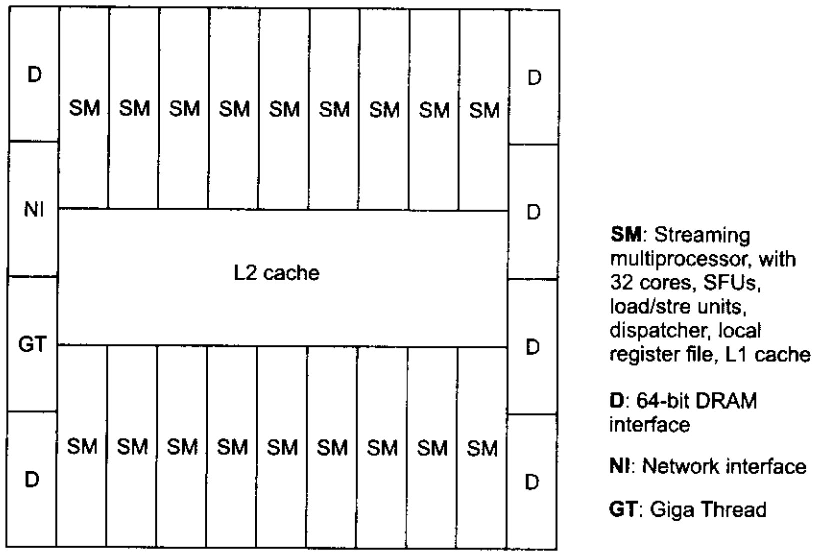

D
SM
SM
SM
SM
SM
SM
SM
SM
SM
D
NI
D
L2 cache
GT
D
D
SM
SM
SM
SM
SM
SM
SM
SM
SM
D
SM: Streaming multiprocessor, with 32 cores, SFUs, load/stre units, dispatcher, local register file, L1 cache
D: 64-bit DRAM interface
NI: Network interface
GT: Giga Thread

Fig. 13.7 Block diagram of Fermi GPU

When we compare stream processing with other available technologies for achieving specialized and power efficient processing, the following broad picture emerges:

(i) Application specific ICs (ASICs) have comparable performance and are power efficient, but they involve longer design cycles and design costs, and are less flexible.

(ii) Field-programmable gate arrays (FPGAs) are less energy efficient, and do not allow applications to be programmed in higher level languages.

With these advantages going for them, it is quite likely that we shall see more specialized applications of stream processors in the coming years.

13.3 CASE STUDIES

13.3.1 Cray Line of Computer Systems

The name of Seymour Cray[15] is well-known in computer industry and academia for his path-breaking innovations in supercomputer architecture, including innovative packaging and cooling technologies. The design of earlier Cray vector supercomputers has been described in this book (see Chapter 8).

In the next line of products, Cray computer systems combined multiprocessing with vector processing. Cray X-MP, the first so-called *multiprocessor supercomputer*, has been described earlier in this book. Its more powerful successor Cray Y-MP was also a hugely successful multiprocessor supercomputer.

In the category of massively parallel processing (MPP) systems, Cray came out with T3D and then its more powerful successor T3E, both of which used a 3-D torus topology. The increasing costs of VLSI processor design had by then led computer system architects to opt for proven processor designs of other manufacturers. T3D and T3E both employed different versions of the 64-bit DEC Alpha processors[16].

Subsequently, Cray introduced the XT series of so-called *scalable Linux supercomputers*. This is currently Cray's top-of-the-line massively parallel supercomputer, and further technical details of the system are provided later in this section.

Cray XMT supercomputer, announced in 2006, is a descendent of the Tera/MTA massive multi-threading concepts. The system uses Cray's own 500 MHz, 64-bit Threadstorm processors, each of which can support 128 threads. With as many as 8000 processors, the XMT system can deliver over one million concurrent processing threads; total shared memory on the system, at up to 8 GB per node, can be up to 64 terabytes. The system is designed to provide the very high levels of multi-threading performance needed for applications such as data analysis, data mining, predictive analytics, and pattern matching. The system interconnect used is Cray's proprietary SeaStar technlogy, which is also used in the XT5 and XT6 supercomputers (see below). Scalar processing, I/O and service functions are provided by AMD Opteron-based nodes.

Cray CX1 is a lower-end supercomputer from the company which is less expensive and easier to deploy. It makes use of Intel Xeon processors in a cluster architecture.

[15] Seymour Cray [1925–1996] is known as *the father of supercomputing*, and is close to being a legend in this field. The following statement attributed to him should be of interest to any student of computer architecture: *Anyone can build a fast CPU. The trick is to build a fast system.*
Seymour Cray was the chief designer of CDC 6600, the first commercial supercomputer ever built, at Control Data Corporation. This was followed by CDC 7600, before Cray founded his first company Cray Research, which built Cray 1 and Cray 2. That first company has since undergone several corporate takeovers and makeovers, and is presently established as Cray, Inc. See ***http://www.cray.com.***

[16] DEC stood originally for Digital Equipment Corporation, of Maynard, Massachusettes, which was at one time the world's second largest computer company. It was taken over by Compaq, and at a later stage that company became part of HP. See *http://www.hp.com.*

In modern computer systems, power consumption and packaging play a key role in system design. Packaging determines not only the length that signals have to travel, but also the aggregate cooling requirements of the system. Since the days of Cray 1, Cray computer systems have been justifiably well-known for technical innovations in system architecture, packaging and cooling.

A recent initiative in the Cray line of systems is the concept of *adaptive computing*—the idea being to adapt a *hybrid* parallel processing computer system to each application through innovative software. The word *hybrid* here refers to a system which combines the elements of vector processing, parallel processing, and multi-threading.

Since Seymour Cray developed the earliest supercomputers, almost forty years ago, we can discern in the history of successive Cray products the broad direction in which computer technology and architecture have moved since those early days.

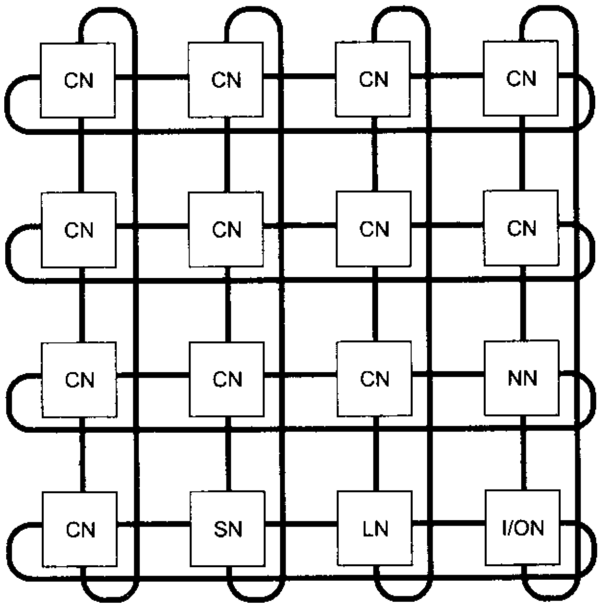
Cray XT Supercomputers The major current ranges of Cray supercomputers—at present capable of reaching petaflops performance—are in the XT series, and in particular XT5 and the recently announced XT6. XT5 supercomputers have reached sustained petaflops performance; one particular XT5 system at Oak Ridge National Laboratory in the US (nicknamed 'Jaguar'), is currently rated as the world's most powerful supercomputer. That particular system uses six-core AMD Opteron processors, with a total of over 224,000 processing cores in the system, and can reach peak performance of over two petaflops.

The brief description given below is specific to XT5, but it also serves to introduce Cray's present supercomputer technology. The main goals of this technology are high computing performance with scalability and programmability. At the same time, advanced packaging, efficient cooling, and low power consumption have all along been characteristic features of Cray products; the OS platform employed is Linux-based. XT5 is based on AMD Opteron processors (quad-core or six-core) in a 2D torus network which is built using Cray's proprietary SeaStar interconnect.

Each diskless *compute node* in the network is made up of two Opteron processors, which have a shared 25.6 GB/sec data path to shared local memory; the local memory is 16 GB or 32 GB DDR2 memory provided with ECC. Each processing core has 64K L1 instruction cache, 64K L1 data cache, and 512 KB L2 cache, and in addition the processor chip provides 6 MB shared L3 cache.

The proprietary SeaStar high-bandwidth interconnect is based on HyperTransport physical links. Each SeaStar ASIC (*application specific IC*) chip is provided with four 12-bit wide network links to the four neighbouring SeaStar chips in the 2D torus, and one link to the node itself. Each inter-node link, provided with specially-designed link-level software, has peak bi-directional bandwidth of 9.6 GB/s and sustained bandwidth in excess of 6 GB/s. Dimension order routing and, for reduced latency, *virtual cut-through* are used within the built-in high-speed routers. The chip also has a direct memory access (DMA) engine, a communication-cum-management processor, and a service port.

As seen in Fig. 13.8, the 2D torus network in the system can be configured with the required combination of compute nodes, I/O node(s), network node(s), login node(s) and system node(s). Storage arrays are connected to I/O nodes, being scalable with the number of I/O nodes provided; the file system manages striping of file operations across the storage arrays. The network node(s) provide Gigabit Ethernet, 10 Gigabit Ethernet, Fibre Channel (FC), and InfiniBand connections.



CN:	Computer Node
SN:	System Node
LN:	Login node
I/ON:	I/O node
NN:	Network node

Fig. 13.8 Schematic of 2D torus network in Cray XT5

For its XT supercomputers, Cray has developed its own Linux-based *Cray Linux Environment* (CLE). The OS kernel operating at compute nodes can be configured for different workloads. For custom applications in which performance and scalability are of primary importance, the compute nodes can be run in a lightweight kernel mode, i.e. with a very thin OS layer intervening between the custom application and hardware. When running standard applications, for which compatibility may be more important, the compute nodes can be configured with a compatible Linux layer, which provides the OS services needed for application compatibility. A single-root file system is maintained across all nodes, which can be inter-operated with other files systems such as NFS.

Program development software supported includes Fortran 90, Fortran 95, C, C++, MPI 2, Cray's shared memory software SHMEM, OpenMP (used within a single compute node), and high-performance math libraries. Other supporting software provided on the system includes performance analysis tools which assist in achieving better resource utilization and load-balance. Application programs developed for XT can be based entirely on MPI, with each core in the compute nodes running an MPI task; alternatively, OpernMP can be used within compute nodes and MPI across compute nodes.

Other critical supporting hardware and software features provided on the system include system monitoring, fault identification and recovery, checkpoint and restart, system interconnect management, system status displays for the administrator, redundant power supplies and voltage regulator modules, and redundant data paths to the system RAID.

At the 2009 Supercomputing Conference, in November 2009, Cray announced its high-end XT6 supercomputer system, which employs eight- and twelve-core AMD Opteron processors to provide higher processing performance than XT5; each compute node in XT6 can be provided with 32 GB or 64 GB of ECC DDR3 local memory. In future systems, XT6 can be upgraded to 12- and 16-core Opteron processors.

Both XT5 and XT6 supercomputers are also available in fully-compatible midrange versions XT5m and XT6m respectively.

13.3.2 PowerPC Architecture, IBM Power7 & Blue Gene

PowerPC Architecture The first articulation and implementation of the concept of reduced instruction set computing (RISC) is believed to have been by a team led by John Cocke at IBM[17], the resulting processor being known as IBM 801 processor. This processor later evolved into IBM's Power processor architecture.

In the early 1990s, IBM, Apple and Motorola[18] used the Power architecture as a basis to define the PowerPC architecture, with the letters PC denoting *performance computing*. PowerPC has a simpler instruction set architecture than the earlier Power architecture; in this sense PowerPC architecture is more in the spirit of RISC and facilitates high performance implementations.

PowerPC processor architecture has been designed for a very broad range of applications—from low cost applications, such as embedded applications, to very high performance systems with multiple processors. Any designer and builder of a specific PowerPC compliant processor must therefore select the target range of applications. About a dozen companies currently produce processors in this family, IBM being one of them.

PowerPC architecture includes compatible 32-bit and 64-bit operating modes. Functional partitioning within the processor makes it suitable for providing superscalar capability; the design aims at maximizing processing throughput rather than clock speed.

There are by now dozens of processors in the PowerPC architecture family; these processors are designed for various different applications—as embedded processors, in game consoles, in servers and mainframes, in high performance computing systems, and others. As a specific example, we shall take a brief look below at the ambitious Power7 processor currently under development at IBM.

IBM Power7 Processor IBM Power7 is a high performance server processor under development which—when it is released in 2010—is likely to be the most powerful processor in the large PowerPC family. The processor is designed using 45 nm VLSI technology, and has about 1.2 billion transistors on a chip area which is slightly under 6 cm^2. The design clock speed of the processor is slightly over 4 GHz.

Power7 is planned as a multi-core processor with 4-, 6-, and 8-core versions. Each core supports 4-way simultaneous multithreading (SMT). A system will in general consist of multiple circuit boards, each of them with multiple processor sockets.

As we have discussed above, bandwidth becomes a critical requirement in supporting multi-core, multi-socket and multiprocessor systems. To address this requirement, each Power7 processor has a pair of 4-channel DDR3 controllers, to sustain 100 gigabytes per second of memory bandwidth. The large 32 MB L3 cache uses so-called *embedded DRAM* technology for reduced chip area and power consumption.

[17] See *http://www.ibm.com*. The interested reader may see *The Evolution of RISC Technology at IBM*, by J. Cocke and V. Markstein, IBM Journal of Research and Development, 34(1):4–11, 1990.

[18] See *http://www.power.org*. For the purpose of this brief case study, we feel it is not essential to discuss in detail the differences between Power and PowerPC instruction sets.

To support superscalar operations, each Power7 core has twelve functional units: two integer units, four double precision floating point units, two load/store units, and one each of decimal floating point unit, vector unit, branch unit, and condition register unit. The decimal floating-point unit addresses the needs of typical mainframe applications.

Within the processor, after the decode stage, instructions are dispatched and tracked in bundles which occupy six time slots of the processor clock. The idea behind this feature is that the hardware bookkeeping which is required for tracking instructions during their execution is thereby simplified.

IBM Blue Gene IBM Blue Gene refers to a series of massively parallel supercomputers being designed and built mainly by IBM, but with active support from the US Government and academia. Blue Gene/L, the first in this series of supercomputers, has already been delivered at a few sites and has been in operation. One key targeted application of this supercomputer is in carrying out biomolecular simulations to study, for example, the folding patterns of protein molecules, which have a bearing on their function.

Targeted peak performance of this supercomputer is the *petaflop* region, i.e. 10^{15} floating point instructions per second. Since per processor performance is in the range of gigaflops, it is clear that petaflop range performance can only be achieved with a large number of processors operating in parallel.

A computation node in the massively parallel system has two cores of PowerPC 440, with shared on-chip L3 cache. Inter-processor connection network has a basic 3D torus topology, but two other networks are also provided—one for global communication and another for barrier synchronization. Each core in the system runs very lightweight Linux OS with a single process. Processors can be partitioned amongst multiple applications, with the additional benefit of improved fault isolation. It is noteworthy that standard Linux applications such as MySQL have been run successfully on the system.

A single Blue Gene/L cabinet houses up to 1024 computation nodes, while the system can host a maximum of 65,536 (i.e. 2^{16}) computation nodes. Processor clock speed is 700 MHz—kept relatively low for reduced power consumption. Power consumption is an important issue in such MPP systems, since a reduction in power consumption allows denser packaging, and also reduces overall power and cooling demands.

The design of Blue Gene has been recognized for its many technical innovations, and the supercomputer series is likely to provide several landmarks in the development of high performance computer systems. After Blue Gene/L, subsequent and more powerful supercomputers in the Blue Gene series are designated with letters P and Q. An explicit design aim is to achieve higher computing performance *per watt*, and thereby allow systems to be built with larger numbers of processors operating in parallel.

13.3.3 Tilera's TILE64 System

VLSI technology today allows the design and fabrication of chips with over a billion transistors. *System-on-a-chip* (SoC) is now a reality, but the question is how to divide the on-chip resources amongst the functional blocks and the vital interconnects, which extend both within the chip and to external memories and interfaces. We may say that the basic question is how best to *architect* a system. To understand better the design trade-offs involved, we now look at an innovative new architecture of a system-on-a-chip.

Tilera Corporation[19] is co-founded by MIT Professor Dr Anant Agarwal, who is considered a pioneer in developing the system architecture exemplified in TILE64. Earlier, Dr Agarwal was closely associated with

[19] See *http://www.tilera.com*

the Alewife project at MIT, a scalable multiprocessor system based on *cache coherent non-uniform memory access* (ccNUMA) design, making use of single chip processors.

TILE64 is a 64-core processor for embedded applications, in which each chip consists of a regular 8×8 grid of *tiles*. Typical embedded applications for which TILE64 is well-suited are those which are very highly computation-intensive, such as network routers, encryption/decryption, video applications, and signal processing.

As seen in Fig. 13.9, *each tile* on the TILE64 chip has its own general purpose processor core, L2 cache, and a non-blocking mesh router to provide for communication with other tiles on the chip, and for off-chip data traffic with main memory, I/O devices and networks. The name given by Tilera to this on-chip mesh interconnect is iMesh.

TILE64 is fabricated using 90nm VLSI technology, and runs at speeds of up to 900 MHz.

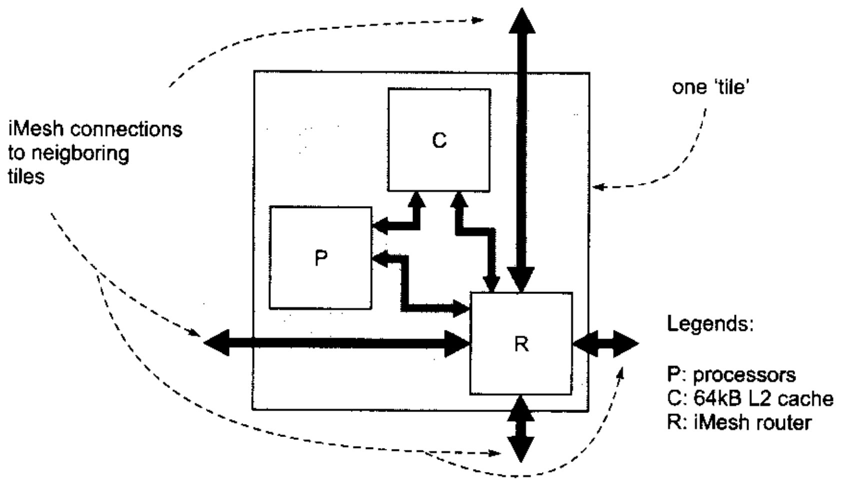


Fig. 13.9 One tile in the TILE64 system-on-a-chip

The processor core in each tile is a relatively simple RISC-style processor which does not, for example, provide for out-of-order execution of instructions. Each processor core has three functional units: two 32-bit integer ALUs and a load-store unit. The design emphasis in TILE64 is not so much on the processor, but on the iMesh-based system-on-a-chip architecture, cache management, support for high data rates required to main memory, and other critical elements which impact *system performance.*

On an L2 cache miss, the processor checks the other on-chip L2 caches for the needed data, before making a slower access to main memory. In this sense, the combined L2 cache memories of the 64 processors can be viewed as forming an L3 cache.

According to Tilera Corporation, "iMesh provides each tile with more than a terabit of bandwidth, creating a more efficient distributed architecture and eliminating the on-chip data congestion". In fact iMesh consists of five independent communication structures, which provide for, respectively:

- Communication between user processes/threads running on tiles
- Communication with I/O devices

- Communication with off-chip main memory
- Tile-to-tile cache transfers
- Low latency interconnect for streaming data

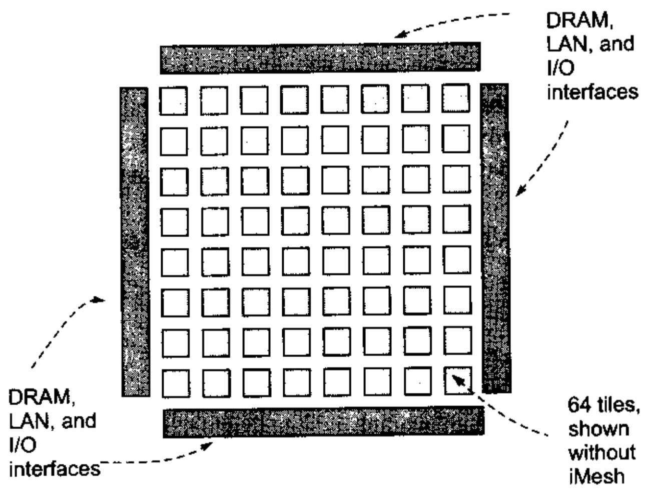

DRAM, LAN, and I/O interfaces
DRAM, LAN, and I/O interfaces
64 tiles, shown without iMesh

Fig. 13.10 TILE64 system-on-a-chip

Figure 13.10 depicts the architecture of this 64-core system-on-a chip.

13.3.4 Sun UltraSparc T2 Processor

Starting from around the mid-1980s, the concept of *reduced instruction set computing* (RISC) began to be more widely known, following the work done by David Patterson at the University of California, Berkeley, and John Hennessey at Stanford University. The work done at Berkeley led subsequently to the development of the Sparc processor by Sun Microsystems[20] in the late 1980s.

The basic idea of RISC is that, with a *reduced* instruction set, a processor can in fact perform more useful work per second. The two key elements of processor architecture which make this possible are the instruction pipeline and the cache memory (which in today's processors may be organized at L1, L2 and L3 levels).

The original Sparc processor is a 32-bit RISC processor with load-store architecture, relatively simple addressing modes, and register-to-register arithmetic/logic machine instructions in three-address format. Separate registers are provided for integer and floating point operands. Of the 32 integer registers, some play a special role in passing arguments during function calls from the calling to the called function.

UltraSparc is the 64-bit enhanced version of Sparc, and UltraSparc T2 is the newest *multi-core, system-on-a-chip* version of UltraSparc with extensive on-chip support for multithreading, networking, I/O, and other key functions.

For increased processor performance, one design option for processor designers is to maximize *instruction issue rate* by increasing the number of stages in the instruction pipeline. The idea is that—with each

[20] See *http://www.sun.com*. Before Sparc, the company used Motorola's 680x0 series processors in its workstations.

pipeline stage doing relatively less work in a clock cycle—it is possible to drive the processor with a higher clock frequency. However, the problems of pipeline flushes and stalls do not go away, and the total power consumption of the chip increases rapidly with clock frequency. As a result, total power consumption of the chip becomes a limiting factor in achieving higher performance.

In this connection, the following observation is of interest[21]:

Power and memory latency considerations place additional obstacles to improving single-thread performance. While recent attempts at improving single-thread performance, through even deeper pipelines, have led to impressive clock frequencies, these clock frequencies have not translated into demonstrably better performance over less aggressive designs.

UltraSparc T2 and its predecessor UltraSparc T1 are designed to achieve higher processing throughput by adopting a different strategy. The architecture of these multi-core chips is designed for those highly demanding applications which exhibit a large degree of thread level parallelism (TLP), but not necessarily much instruction level parallelism (ILP). The strategy implies that the compute time and memory latencies of multiple executing threads are interleaved in time, with increased total throughput.

UltraSparc T2 has eight processor cores on the chip, with each supporting eight-way fine-grained multi-threading. Overall, therefore, the chip supports sixty four parallel threads. The chip also contains a crossbar switch, shared L2 cache, and extensive support for I/O and networking, and therefore it is in fact a *system-on-a-chip* (SoC). Since the threads run independently of each other and share hardware resources, each thread behaves as a processor in its own right; thus the single chip can support 64 *virtual systems*.

Figure 13.11 depicts schematically the architecture of UltraSparc T2.

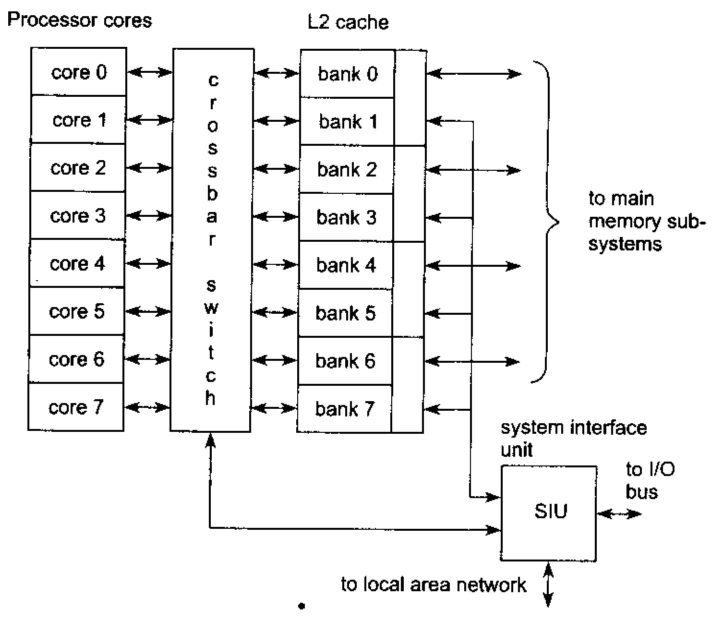


Fig. 13.11 Architecture of UltraSparc T2 system-on-a-chip

[21] *From Chip Multithreading: Opportunities and Challenges*, by L. Spracklen and S. G. Abraham of Sun Microsystems, IEEE International Symposium on High-Performance Computer Architecture (HPCA-2005), 2005.

The T2 chip has an area of just under 3.5 cm^2, with about 500 million transistors on it, and is fabricated using a VLSI process of 65 nm line width. The chip can operate at 1.4 gigahertz with 1.1 volt supply, and has 1831 pins on its underside for connection to the rest of the computer system. Nominal power consumed by the chip is 95 watts; on the basis of power per cm^2 of area, this is greater than that of an iron. However, on a per thread basis, this power consumption works out to be quite low.

Each of the eight processing cores on the T2 chip has its own data paths, register sets for multiple threads, two integer operation units, and a floating point unit. In addition, each core has hardware provided for cryptography and graphics, and has support for eight-way fine-grained multi-threading. These hardware resources, and the *per core* L1 instruction cache and data cache, are shared by eight threads executing on each core.

At the level of the system-on-a-chip, other hardware resources such as the 4 MB L2 cache are made available to the eight cores by means of a crossbar switch. For faster access, the L2 cache is organized in the form of eight parallel banks. Memory interfaces, I/O interfaces, and networking are also shared amongst the cores. Networking capability consists of two network interfaces of 10 gigabits/second each.

Design of UltraSparc T2 is targeted towards compute-intensive applications with a high degree of multi-threading. Apart from back-end servers, these include network devices such as packet routers, switches for local area networks, graphics and imaging applications, and other similar applications.

A unique feature of the UltraSparc T2 chip is that its complete design has been made available on the web to researchers and developers under an 'open source' arrangement. The stated objective behind this decision by Sun Microsystems is to encourage innovations around the world in processor design and applications.

13.3.5 AMD Opteron

AMD[22], a major manufacturer of semiconductor devices, is known for its processors which are instruction-set compatible with Intel's x86 family of processors. Opteron is a high performance 64-bit processor from AMD which maintains instruction set compatibility with the 32-bit x86 instructions without any performance penalty.

AMD's 64-bit instruction set architecture provides for 64-bit operands, 128-bit operands, and 64-bit virtual addresses. The development of such a 64-bit extension of the Intel x86 architecture is targeted towards applications which require huge amounts of memory, examples being high performance servers, workstations, database management systems, and engineering design tools. The processor also provides integer and floating point vector operations for graphics/multimedia types of functions, some of which are known as *streaming SIMD extensions* (SSE); in this category, combined *multiply-add* as well as matrix operations are provided.

Opteron is characterized by a fairly large split L1 cache, with 64 KB for instruction cache and 64 KB for data cache. L2 cache is either 512 KB or 1 MB, depending on the model, while the shared L3 cache goes up to 6 MB on the newest six-core models. Since 2003, when the processor was introduced, it has been implemented using the successive 130 nm, 90 nm, 65 nm and 45 nm VLSI technologies.

The processor architecture is 3-way superscalar—i.e. up to 3 instructions can be completed per clock cycle. *Speculative* and *out-of-order execution* is provided, as is *register renaming*, to remove apparent dependencies between instructions in the instruction pipeline.

[22] See *http://www.amd.com*.

Support for multiprocessor systems is provided on the basis of *cache coherent non-uniform memory access* (ccNUMA), rather than the symmetric multiprocessing (SMP) design with a common shared memory. A processor can access the memory of another processor, and sophisticated snooping hardware is provided to ensure cache coherence. Compared to SMP systems, such systems can support a higher degree of multiprocessing without running into the memory bandwidth bottleneck.

Unlike in the Sun UltraSparc T2, there is no hardware support in the Opteron processor cores for multithreading, reflecting the different design objectives behind the two processors. Multi-core versions of the processor have been manufactured with up to six cores per chip. HyperTransport links (see Section 13.1.4) are used both for processor-memory communication and for inter-processor communication.

As seen above, Cray has incorporated the Opteron processor into their XT series of supercomputers, which can support configurations having thousands of processors. Some of the world's most powerful supercomputers are based on this Cray architecture using large numbers of Opteron processors. Sun Microsystems uses the Opteron processor in its high-end servers having large processor counts.

When we compare the design of the Opteron processor with those of TILE64 and UltraSparc T2 processors (see above), a natural question arises with reference to the basic design goals of any computer system:

Which of the following two models of computation should the system architecture target?

(i) A relatively smaller number of 'heavier' threads of computation, with one thread running per core, or
(ii) A larger number of 'lighter' threads of computation, with multiple threads running per core.

Clearly, the choice depends on the class of applications for which the system architecture is being designed. Opteron, UltraSparc T2 and TILE64 represent different possibilities in the number of execution threads *versus* the processing power per thread. The architects of a computer system, knowing the targeted application load, must make the right choice.

13.3.6 Intel Pentium Processors

Intel[23] 8088, used in the original IBM PC, was a 16-bit processor with 20-bit physical address, i.e. total physical address space of 1 megabyte. Logical memory space consisted of four segments—namely, *code*, *data*, *stack* and *extra* segments. A 16-bit segment offset meant that each segment was limited to 64 kilobytes.

As VLSI technology advanced, successively upgraded members of the so-called x86 processor family—80286, 80386 and 80486—had larger memory address space, 32-bit word size, higher clock frequencies, on-chip cache and memory management functions, and additional instructions, including floating point instructions. Successive models of the immensely popular PC were built around these processors.

Therefore, maintaining *backward compatibility* of instruction set with earlier processors of the x86 family has always been a non-negotiable design requirement of any new processor of the family, since all software written for earlier versions of the PC had to run with its subsequent versions. This critical business requirement pushed Intel processor designers to the limits of their ingenuity—since they had to achieve higher processor performance with every model, while maintaining at all times full backward compatibility of instruction set.

With rapid advances in VLSI technology, as it became possible to build enormously more powerful single-chip microprocessors, the spotlight turned on the critical role of the instruction set in achieving high

[23] See *http://www.intel.com*

performance. Benefits of the RISC approach soon became clear to processor designers, and all new processor designs benefited from the new ideas. But the only way for the Intel x86 family of processors to maintain backward compatibility in instruction set was to continue with its CISC approach.

Designers at Intel pushed the frontiers of VLSI technology to achieve higher processor performance while maintaining backward compatibility. The original Pentium and its successors were introduced as advanced sequels to the Intel 80486 processor. Even with the inherited CISC instruction set, these processors combined standard RISC design techniques in their internal architecture—such as a *micro-operation pipeline*, multiple functional units, and out-of-order sequencing.

Figure 13.12 is an overview of the architecture of the Pentium 4 processor.

The processor has two levels of cache memory—L1 and L2. The faster but smaller L1 cache is divided into 8 kilobytes of instruction cache and 8 kilobytes of data cache, while the larger L2 cache is a combined instruction and data cache of 256 or 512 kilobytes, depending on the processor model. The main memory may also be provided with an additional off-chip L3 cache.

The Fetch/Decode unit (marked 'A' in the figure) is connected to the L1 instruction cache. This unit fetches and decodes successive instructions, producing several so-called *micro-operations* corresponding to each machine instruction. These micro-operations are forwarded to the *micro-operation buffer* (marked 'B'), in which micro-operations produced by multiple machine instructions are buffered.

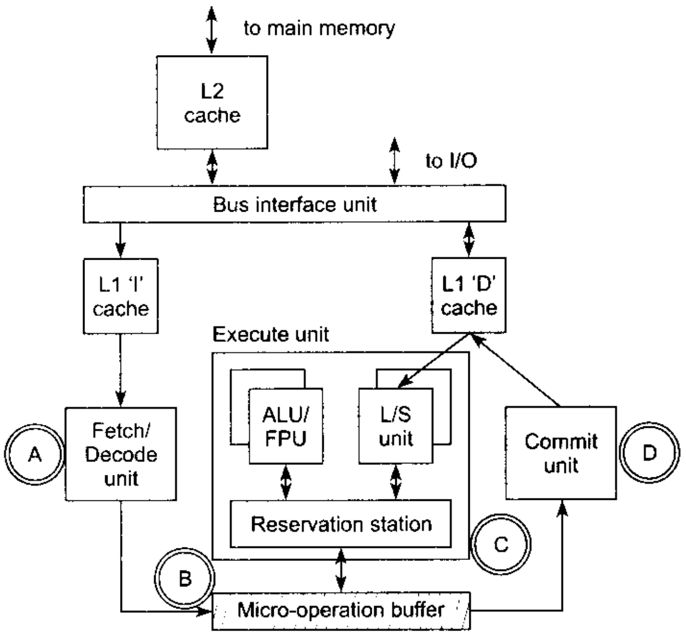


Fig. 13.12 Internal architecture of the Pentium 4 processor

For execution by specific functional units—such as the integer ALU or the FPU—micro-operations are forwarded to a *reservation station*. An operation is performed in a unit when its operands become available and the functional unit becomes free. Execution of micro-operations need not follow the order in which machine instructions are fetched. Data load and store operations on memory are carried out by the *load unit* and *store unit*, respectively, which operate as functional units connected to the L1 data cache.

The reservation station and the functional units together make up the *execute unit* of the processor (marked 'C'). Within this execute unit, hardwired control is provided for the simpler instructions of the processor, whereas complex instructions of the CISC type are provided with microprogram control. This is one of the ways in which RISC and CISC approaches have been combined in the internal architecture of the processor.

When all the micro-operations of an instruction have been performed in the execute unit, the instruction is *committed* (or *retired*) to main memory. This work is carried out by the *commit unit* (marked 'D'), which ensures that completed machine instructions are committed in the order in which they are fetched, as the programmer expects.

Fetch/decode unit ('A'), *execute unit* ('B') and *commit unit* ('D') operate in parallel, sharing the common *micro-operation buffer* ('C'). Thus these three units can be said to form a 'high-level' pipeline through which instructions pass. But each of these units is also implemented as a pipeline, so that multiple instructions can be in each of these units at one time, each in a different stage of processing. Branch prediction logic, which is required with the instruction pipeline, is also provided.

Memory management functions on the processor provide support for virtual memory using paging and/or segmentation, as well as memory protection for user programs and the operating system; segments may be shared between running programs.

13.4 PARALLEL PROGRAMMING MODELS AND LANGUAGES

With all the recent advances in the hardware architecture of high performance computer systems—of which we have seen a few examples above—it is still a major challenge to map an application program to make efficient use of the underlying hardware. Inability to achieve this aim results in the gap between theoretical peak performance of a system and the actual application performance achieved in practice.

To allow software designers to build parallel applications, one possibility is to provide so-called *parallel constructs* as extensions to sequential higher level languages. Chapters 10 and 11 of the book discuss some of the relevant issues in this context.

But another exciting possibility is to design a new parallel programming language from first principles, to provide parallel programming constructs which are based naturally on the way in which parallel algorithms are conceived. We shall now see an example of this approach, a new parallel programming language being designed at Cray.

13.4.1 Parallel Programming Language Chapel

Chapel is a new parallel programming language being developed by computer scientists at Cray Research. The word *Chapel* has been derived from *cascade high-productivity language*—by taking the first letter of each of these words and inserting a couple of helpful vowels. The project is currently at research and development stage, in which several Universities and research centers are collaborating. The Chapel developer team clearly indicates that it is open to work with other computer scientists interested in parallel programming.

Chapel is one of three such parallel programming languages being developed in the US under the prestigious High Productivity Computing Systems (HPCS) program, the other two being X10 being developed by IBM, and Fortress by Sun Microsystems[24].

The specific goals behind the development of Chapel are: programmer productivity, programmability of parallel computers (improving over the older parallel programming models), better portability, and robustness of the parallel programs developed. Target architectures for the parallel machine language programs generated using Chapel are multi-core systems, computing clusters, as well as special high performance computing platforms from Cray and other vendors.

Chapel is intended for general parallel programming; it provides high level abstractions for data-parallelism, task-parallelism, and nested parallelism. The aim is that the language should allow all the broad types of software parallelism to be expressed, and should be targeted towards general levels of hardware parallelism.

Chapel provides *global-view abstractions*, i.e. program structures which allow the source code to describe the computation as a whole—rather than parallel fragments (such as processes or threads) that must be somehow made to communicate and work together. The language provides for so-called *multi-resolution design*, which means that the programmer has the choice of using higher level or lower level abstractions. Control of locality is provided, so that data and computational threads can be placed at specific locations within the parallel processing system.

Programs written in a language such as Chapel do not overtly depend on MPI (or similar) library of communication or synchronization functions, since the semantics of parallel computations are provided in the language itself.

Overt use of lower level functions in a program—such as those in MPI—does not hide communication and synchronization mechanisms; such lower level functions thereby complicate programs and make them error-prone and difficult to maintain. Compared to Chapel, parallel programming using MPI functions can therefore be thought of as analogous to assembly language programming.

Language Features

Like C or JAVA, Chapel is a block-structured, typed language with imperative statements. Object-oriented programming features are provided—similar to those in C or JAVA—but Chapel programs can also be written without using these features; explicit manipulation of pointers is avoided. Parallel programming features in Chapel are based on the features earlier introduced in ZPL, HPF[25] and Cray's own parallel version of C/ Fortran developed for their MTA systems.

(a) Data Parallelism Data parallelism is supported with the use of *domains*, distributed domains and arrays, and iterators based on index sets. In the example below, D is being defined as a 2-dimensional domain with integer index values running from 1 to 4 and 1 to 8, respectively.

```
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
```

[24] For more information, see http://chapel.cray.com, http://x10-lang.org, and http://projectfortress.sun.com

[25] See *http://www.cs.washington.edu/research/zpl* and *http://hpff.rice.edu/*

Now one or more arrays, with elements of any base type, can be defined using D as the underlying domain which defines the array structure. For example:

```
var A, B: [D] real; // Note use of domain name D
```

Thus an array is created when each element of the underlying domain is mapped to a data element of the base type, which is `real` in the example above. A sub-domain of a domain can also be defined, as in:

```
var InnerD: subdomain(D) = [2..m-1, 2..n-1];
var smallA: [InnerD] real;
```

The relationship between domains D and InnerD is clarified in Fig. 13.13.

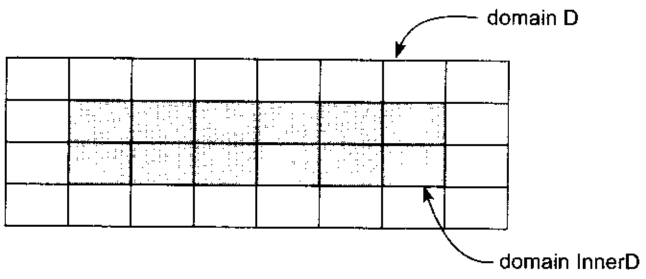


Fig. 13.13 Relationship between domains D and InnerD

Domains or sub-domains can be used to govern iterations, as in:

```
for (i,j) in InnerD do
    A(i,j) = i + j/10.0;
```

A shorthand notation for this same iteration is:

```
[(i,j) in InnerD] A(i,j) = i + j/10.0;
```

Thus domains support data parallelism by defining index sets, based on which arrays can be defined, and in controlling sequential and parallel loop iterations. Both data arrays and loop iterations can be *distributed* across the multiple processing elements in the system.

An *index set* can be of one of three types:

(i) *Arithmetic* indices (seen above) define Cartesian tuples, and are similar to integer indices used in other programming languages; an arithmetic index can optionally be made *strided* or *sparse*.

(ii) *Associative* indices have arbitrary values which serve as keys to hashed structures.

(iii) *Opaque* indices are anonymous, in the sense that nothing is said about the elements making up an index set; such indices support the concept of unordered sets of elements.

Just as functions are defined, *iterators* can be pre-defined to yield successive values from an index set. Such iterators can then be used for governing loop iterations which range over the defined index sets. A pre-defined iterator can be used wherever loops with that particular iterative structure are needed in the program.

(b) Task Parallelism Task parallelism is supported with the use of high level language features as well as by lightweight synchronization operations. Synchronization variables are provided as a special type.

For example, `readFE()` operation on a synchronization variable causes the calling process to wait until the variable is *full*, and makes the variable *empty* after its value is read by the process.

Similarly, `writeEF()` operation on a synchronization variable causes the calling process to wait until the variable is *empty*, and makes the variable *full* after its value is written.

Task parallelism is supported in both structured and unstructured forms. The keyword `begin` initiates a separate thread executing the specified statement, as for example in:

```
begin runMyThread();
```

No join operation is implied when a thread is initiated using `begin`. Structured thread invocation is supported by, among others, the `cobegin` statement, as in:

```
cobegin {
  myThread(1);
  myThread(2);
  myThread(3);
}
```

Note that here an implicit join takes place at the end of the `cobegin` block.

The statement `coforall` executes loop iterations in parallel threads, with an implied join, as in:

```
coforall i in 1..numThreads {
  myThread(i);
}
```

As against this, the statement `forall` implies that iterations may be executed in parallel, depending on the distribution of the concerned domain:

```
forall (i,j) in InnerD do
    A(i,j) = i + j/10.0;
```

Concurrency can be inhibited explicitly, based on conditions specified. *Atomic* operations are supported. An atomic operation is one which, when performed within one thread, appears atomic to all other threads in the program—i.e. they cannot see any partial result produced by the atomic operation.

When a scalar operation or function is applied in parallel to all elements of an array, the operation is said to involve *promotion*. Promotions are executed in parallel, with an implicit `forall` controlling the execution. Given arrays A and B, a simple example of promotion is:

```
B = 2 * A;
```

When an operator is applied to array elements to obtain a scalar value, the operation involves *reduction* (see Section 13.2.2). An example of reduction in Chapel is:

```
sum = + reduce A; // find the sum of all elements in A
```

Simple syntactic notation is also provided in Chapel for domain and array *slicing*; the definition of domain `InnerD` above is an example.

As mentioned above, loop iterations are governed by domains or iterators; and, as we know, loops provide huge potential for parallelism. Therefore several statements are provided in Chapel to specify the mode of loop parallelism to be utilized.

As in C or JAVA, a `for` loop generates a single thread to execute all loop iterations. A `coforall` loop generates a separate thread for each loop iteration. The third variant is a `forall` loop, in which some number of threads are created for the loop iterations, as determined by the loop iterator expression, or by the domain or array distribution.

A special type in Chapel known as *locale* is used to specify an architectural unit of locality of processing; each locale is understood to have processing and memory functionality. In a system, this may refer to a processing element (PE), as we have used the term in earlier chapters, or even a multi-core processor.

For a program running on *N* locales, the locales are numbered from 0 to *N*-1; execution of the program begins with one task running on locale 0. The number of locales available to the program is specified on the command line.

The statement:

```
on loc { statements }
```

causes the specified statements to be executed on the specified locale `loc`. The special locale named '`here`' refers to the locale on which the reference is made.

A *distribution* is a mapping from domain or array indices to locales—i.e. it is the basic mechanism of achieving data distribution across the locales. A ubiquitous variable can be created, with the semantics that each locale has its own copy of the variable, i.e. the variable is replicated on all locales.

Chapel programming language has initially been implemented using Chapel-to-C compilation, followed by standard C compilation and use of support libraries. It is freely available as a download for research purposes and/or for contributing to its further development and refinement.

In the earlier chapters of the book, we have discussed at some length topics such as *compiler-detected parallelism* and *dependence checking within array references in a loop*. It should be noted that, when a global-view parallel programming language such as Chapel is used, the function of the compiler changes substantially. All the data and task parallelism in the program is now made evident in the program at a higher level, and therefore *detection of parallelism* by the compiler is no longer the primary issue. The main goal of the Chapel compiler is to efficiently map the defined parallel semantics of the source program onto the underlying parallel processing hardware.

13.4.2 Function Libraries for Parallel Programming

Standardized functions which support a parallel programming paradigm offer a practical alternative to programming language extensions, because they can work with a range of programming languages, such as C, C++ and Fortran. Since these sequential programming languages have already achieved a *legacy* position in the computing profession, perhaps inevitably the standardization of parallel programming paradigms has become partly separated from issues of language definition.

We now look at several such standardized models of parallel programming, two based on the message-passing model, one on shared memory multiprocessing and one on software multi-threading.

Message Passing Interface (MPI) Message Passing Interface (MPI)[26] is widely used to build applications for distributed memory as well as shared memory architectures. As we know, in the message passing mode of interprocess communication, message data moves from the address space of one communicating process to the address space of the other communicating process, over the underlying communication layers.

The message passing operation requires both the communicating processes to issue appropriate function calls. Typical point-to-point communication under MPI is carried out using the basic send and receive calls *MPI_Send* and *MPI_Recv*; but, as we shall see below, MPI offers much additional functionality as well.

It could be argued that interprocess synchronization and communication achieved through shared memory operations—such as *test-and-set*—is more efficient than message passing. But the fact remains that message passing offers a higher level abstraction for building parallel applications which is more robust against processor speeds, types of interconnects, and so on. Message passing primitives can be provided on both distributed and shared memory architectures, whereas it is not really practical to provide shared memory primitives on a distributed memory system.

MPI supports the general MIMD model of parallel processing, as well as the more restricted *single program, multiple data* (SPMD) version of parallelism. The interface is versatile enough to support a high performance computing platform, a lower cost network of computers, or even modern multi-core chips. Amongst the original design goals of MPI are *source code portability* and *language independence*.

The first version of MPI, known as MPI 1 standard, was published in 1994, supported by a consortium of computer scientists and vendors. It is defined as a specification for a library of functions, available to vendors and other groups for implementation. MPI 2 standard was published in 1998, with provision for additional features such as dynamic process management, remote memory operations, and parallel I/O.

The underlying communication layer for MPI is often TCP/IP, although that is not part of the specification. Given the nature of the message passing mechanism, support for heterogeneous environments is a major natural benefit of the MPI platform.

Apart from basic interprocess message passing and synchronization, MPI provides several additional facilities for the design of parallel applications, such as:

- broadcast, gather and reduce operations
- barrier synchronization between processes
- user-defined topology over the processes
- user-defined data types for C, C++ and Fortran
- synchronous and asynchronous modes of communication
- buffered and unbuffered communication

An MPI application consists of multiple *processes*. Amongst these processes, various modes of communication can be provided using the programming interface which is available through higher level languages such as Fortran, C, and C++. Processes are mapped to hardware processors, which may be on the same chip, the same system, or on different systems which communicate over a network.

[26] See *http://www.mpi-forum.org*

Processes are grouped together into so-called *communicators*; within communicators, messages are sent and received using functions such as *MPI_Send* and *MPI_Recv*.

In Example 13.3 we saw that, with parallel processing, an associative operation—such as addition—can be performed over n operands in $\log_2 n$ steps. In Figure 13.14, for implementing such an operation over eight operands, we see a *logical network topology* in the shape of a *binary tree*. The number of time steps required equals the height of the tree, which is $\log_2 8 = 3$.

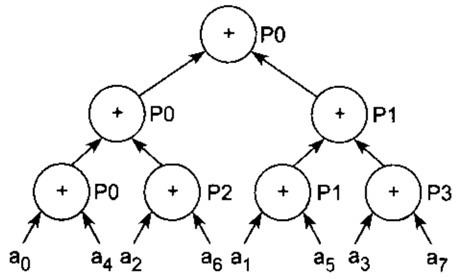


Fig. 13.14 Tree-structured logical process topology

In Fig. 13.14, example process IDs are shown next to the circles which represent processes, while the eight operands shown at the bottom are assumed to reside in the respective processes.

In the first step, four processes are performing the operation; in the second step, two processes are performing the operation; and in the final step, the final result is produced by the process P0 which is shown as the root node. As compared to Example 13.3, here we see the significance of the *logical process topology* in terms of the specific interprocess communication required for a given application.

Let us suppose that the underlying *physical network topology* in this particular case is a 2-D torus (as in Cray XT5). Then the communication pattern indicated above must be achieved through appropriate *routing* over the underlying hardware interconnect. This would be part of the MPI *implementation* rather than its *specification*.

If we count each upward arrow between processes in Figure 13.14 as one unit of communication, then it is easy to see that the total amount of communication taking place, for the operation as indicated over the n distributed operands, is $(n - 1)$. The number of time steps required is $\log_2 n$. Therefore the amount of communication taking place per unit time is proportional to $n/(\log_2 n)$—i.e. it grows with increasing n.

This example clarifies further why very sophisticated inter-processor communication needs to be provided on modern multiprocessor systems or multi-core chips, such as the Cray XT, IBM Blue Gene, and the TILE64 multi-core chip (see above).

openMP openMP—which stands for Open Multi-processing—is a standard API for parallel applications based on the shared memory model of multiprocessing[27]. As in the case of MPI, this standard is also defined by a consortium of computer scientists and vendors. Like MPI, openMP is also a *specification*, for which any multiple computer vendors or other groups can provide compliant *implementations*.

Achieving *portability* and *scalability* in shared memory parallel applications is a major aim of defining the openMP standard. The first version of the standard was published in 1997, and the current version was

[27] See *http://www.openMP.org*

published in 2008. Language support defined within openMP includes C, C++ and Fortran. openMP makes use of *compiler directives* which, if ignored, result in sequential execution of the underlying program. Applications can be parallelized incrementally, and the granularity of parallelism may be coarse or fine.

The basic concept in openMP is that a *master thread* can generate so many *slave threads*, which may be executed in parallel over available processors. Thus the basic parallel construct is a paired combination of *fork* and *join* (see Chapter 10), with an implicit barrier at the point where the slave threads join. This basic parallel construct may be nested, as shown in Figs. 13.15 and 13.16. There is no restriction that the number of threads must equal the number of available processors.

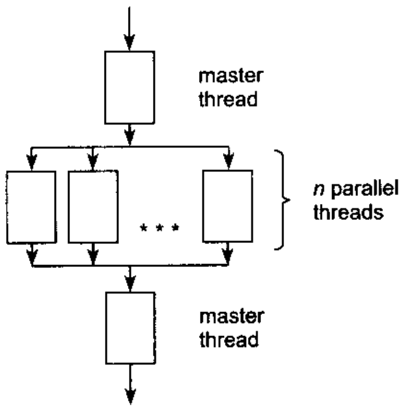


Fig. 13.15 Master thread, slave threads, and implicit barrier

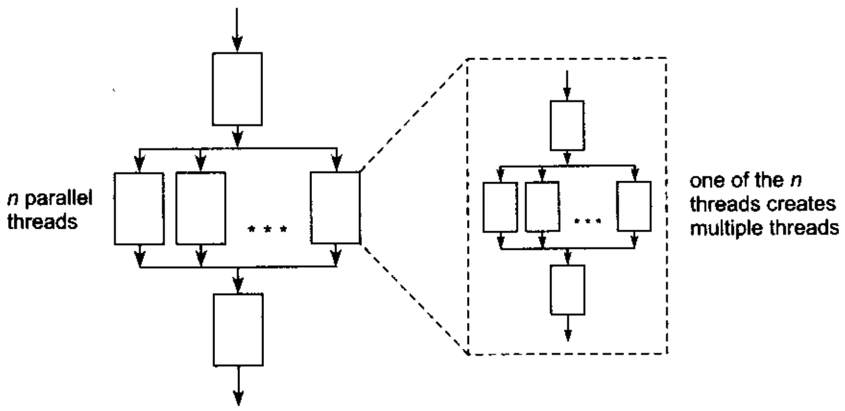


Fig. 13.16 Nesting of parallel constructs

Thread memory may be made private or shared, and there is a flush operation available for shared memory. A feature known as *work-sharing* allows assignment of independent loop iterations to separate threads. Synchronization mechanisms such as critical section and explicit barriers are available; *reduction* operation is also provided.

PThreads POSIX, or Portable Operating System Interface for UNIX, is an operating system interface standard of IEEE which is supported by a large number of computer companies. PThreads—or POSIX Threads—is the part of POSIX which pertains to the development of multi-threaded applications.

Functions and APIs are provided under PThreads for:

(1) Thread management—i.e. create and join threads, set and query thread attributes, etc.
(2) Mutex (mutual exclusion) variables—i.e. create, destroy, lock and unlock operations, used for synchronization amongst threads,
(3) Condition variables—to provide wait and signal communication between threads, and
(4) Synchronization—to provide read and write locks, and barrier synchronization.

Recall that under UNIX, threads exist within a process and use resources allocated to the process by the operating system. Each thread has its independent thread context—PC, registers, thread status, etc.—but all threads share the same process memory image. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads. Processing load may be divided amongst threads in a hierarchical model or a flat peer-to-peer model.

Using PThreads functions, the programmer must provide the required synchronization between threads. Because of lower overheads in managing threads as compared to processes, multi-threading under Unix is much more efficient than using multiple processes. Multithreading using PThreads is also more efficient than MPI on the same processing element, or on a symmetric multiprocessor, because the multiple threads execute out of the common shared memory of the process.

PVM (Parallel Virtual Machine) Parallel Virtual Machine (PVM) is a platform for distributed applications developed at Oak Ridge National Laboratory in the US, in association with other Universities, in the late 1980s and early 1990s. The development was carried out as part of a larger research project into distributed computing.

Under PVM, an application is conceived as a collection of *tasks* (in fact processes) which run in parallel, on one or more machines, and communicate by sending and receiving messages amongst themselves. Tasks are identified by task IDs, and there is also provision for defining groups of tasks.

Heterogeneous processing environments are supported under PVM. A system seen as a *virtual processing element* may be a single-processor system, a multiprocessor, a cluster, or any other type of processing resource. The network underlying PVM may also be heterogeneous, in the sense of being made up of different types of links. Machines—i.e. virtual processing elements—can be added or removed during operation of the PVM system as a whole.

PVM can support *functional parallelism*, *data parallelism* or a combination of the two, using C, C++ and Fortran languages. In many ways, PVM is similar to MPI; but, as we have seen in the case studies, MPI has gained much wider acceptance amongst the community of users who develop parallel applications.

Summary

Major trends and developments in computer architecture are influenced strongly by (a) advances in underlying technology, and (b) growth in range of applications. We started the chapter with a brief

review of some of the key technological advances which have had an impact on processor and system architecture over the last couple of decades.

Steadily decreasing line widths and faster clock speeds have characterized VLSI technology. Graphics processors and displays have become far more sophisticated, giving rise to major new applications such as animation and multimedia. Magnetic disk storage densities and capacities have seen huge increases, even as their cost, size and power consumption have been falling. And, with advances in electronics, signal processing, and underlying communication technologies such as fiber optics, truly revolutionary advances have been seen in system interconnect and network technologies. HyperTransport, PCI Express, Gigabit Ethernet and 10 Gigabit Ethernet are specific examples.

The types of parallelism present in a program can be divided broadly into *structural parallelism* and *instruction level parallelism* (ILP). Processor design has often been focused on exploiting ILP, but system design—of a multiprocessor system, for example—has to be based on the type of structural parallelism present in the target applications. In this sense, the designer must choose between more aggressive exploitation of ILP *versus* a larger number of processor cores with less aggressive exploitation of ILP.

The discussion of parallelism was continued with a simple example, followed by a discussion of work done by a parallel algorithm, work-efficiency, efficient parallel algorithms, and Brent's theorem. Stream processing is a newer form of parallelism which can be exploited when the target application—e.g. graphics, image or signal processing—involves the processing of large amounts of data in the form of data streams. This type of parallelism has characteristics of SIMD as well as data flow processing.

Many innovative high performance products—processors and systems—have emerged in recent years, aimed at different target applications. Several representative products were discussed in this chapter in the form of case studies. These included the Cray line of computers systems and the Cray XT5 system; PowerPC processor architecture, IBM Power7 processor and IBM Blue Gene supercomputer; Tilera's TILE64 system-on-a-chip, Sun UltraSparc T2 processor, AMD Opteron and Intel Pentium processors.

The parallel programming language Chapel, being developed by Cray under the prestigious HPCS program in the US, was reviewed as an example of a global-view parallel programming language. For use in building parallel applications, conventional programming languages such as C, C++ and Fortran need the support of function libraries for interprocess communication and synchronization. Such libraries use either shared memory or message-passing models; as specific examples, the salient features of MPI, openMP, PVM and PThreads were discussed.

Exercises

Problem 13.1 Explain in brief the meaning and significance of *line width* in VLSI technology. What are the various line widths currently being used?

Problem 13.2 State and explain in brief Moore's law. Justify this empirical law in brief, arguing on the basis of development of newer VLSI fabrication technology. Does your justification explain the ratios of the line widths listed in answer to Exercise 1?

Problem 13.3 Faster clock speeds become possible with advances in VLSI technology. What is the effect of faster clock speeds on power consumption? Why is this an important issue in the design of processors and computer systems?

Problem 13.4 Assume that a single processor chip in a parallel system consumes 50 watts of power, and that the system contains 1000 such processors. Assume also that all other components in the system consume, in aggregate, as much power as the 1000 processors. What is the total power consumption of the system, in kilowatts? How many domestic irons, operating together, will dissipate this much power?

Problem 13.5 For the system described in Exercise 4, assume that air-conditioning and lighting consume as much power as the computer system itself, and that the cost of electric power is Rs. 6/- per kilowatt-hour. What is then the monthly cost of electric power for operating the system around the clock?

Problem 13.6 In recent years it has been seen that, beyond a point, processor performance does not increase in proportion with clock speed. List some of the trends in processor design resulting from this basic factor related to VLSI technology.

Problem 13.7 What is off-chip inter-connect delay? What is its significance in system design?

Problem 13.8 With advances in VLSI technology, the total design cost of a VLSI processor has increased enormously. What has been the impact of this increase on the way parallel processing systems are designed?

Problem 13.9 Over the last two decades, with advances in VLSI technology, processor speeds have been increasing much faster than main memory speeds. What has been the impact of this trend on computer architecture?

Problem 13.10 A hand-held computing device has to be provided with color graphics display having a resolution of 300×400 pixels, with $3 \times 8 = 24$ bits of color information per pixel. The quality of animation to be provided requires refresh rate of 30 frames per second, and an average of 2 arithmetic operations are required per frame per pixel. Calculate the graphics processing power needed, in million arithmetic operations per second, for this particular application.

Problem 13.11 List the three different ways in which multiple disks can be used in combination in a RAID system, and the corresponding benefits expected in terms of the performance of the storage system.

Problem 13.12 In parallel processing systems with multiple processors, there has been a trend away from shared media interconnects to switched media interconnects. Explain briefly the reasons behind this trend.

Problem 13.13 List the salient technical characteristics of HyperTransport interconnect technology, and describe in brief its possible application in a multiprocessor system.

Problem 13.14 List the salient technical characteristics of Low Voltage Differential Signaling (LVDS), and the performance benefits which it provides.

Problem 13.15 Contrast the salient characteristics of PCI and PCI Express interconnect standards.

Problem 13.16 List the salient technical features of Gigabit Ethernet, and explain in brief the meaning and utility of the concept of Qualty of Service (QoS).

Problem 13.17 Describe in brief the concept, applications and benefits of cluster computing.

Problem 13.18 What do you understand by *structural parallelism* in a parallel program? List the different possible forms of structural parallelism which we have studied. Contrast this concept with *instruction level parallelism* (ILP), and discuss whether there are any trade-offs involved in processor and/or system design between supporting these different types of parallelism.

Problem 13.19 Define in brief the meaning of *virtualization*, and explain why this concept is closely related to hardware support for multi-threading.

Problem 13.20 Example 13.2 discusses a two-dimensional numerical integration. Outline how you would extend this concept to three-dimensional integration, wherein a function *f*(*x*,*y*,*z*) of three variables is integrated over a volume of integration defined by limits Xmin, Xmax, Ymin, Ymax, Zmin and Zmax along the three axes X, Y and Z respectively. Justify the number of processors used in various phases of the algorithm.

Problem 13.21 Define the work performed by a parallel algorithm. When can we say that a parallel algorithm is work-efficient with respect to another algorithm? When is a parallel algorithm optimal?

Problem 13.22 Two versions of matrix multiplication algorithm are shown in Example 1.5. Justify these two versions as either being or not being optimal.

Problem 13.23 When do we say that a parallel algorithm is efficient? Are the matrix multiplication algorithms of Example 1.5 efficient?

Problem 13.24 State Brent's theorem. Explain in brief its significance and underlying assumption(s).

Problem 13.25 List the characteristics of stream processing using a schematic diagram of processing kernels and data streams. List three typical applications of stream processing, and the stated goals of the research project Merrimac at Stanford University.

Problem 13.26 Draw a block diagram and list the salient features of the Fermi stream processor introduced by Nvidia. List the advantages of using a stream processor as against using ASIC(s) and FPGA(s) for a given application.

Problem 13.27 Describe in brief the salient features of:

(a) Cray XMT supercomputer
(b) IBM Power7 processor
(c) IBP Blue Gene supercomputers.

Problem 13.28 State in brief the salient features of Cray XT5 supercomputer; draw a schematic diagram of the 2D torus system interconnect, showing the different types of nodes which are connected.

Problem 13.29 Recall that PowerPC defines a processor architecture rather than a processor itself. List the salient features of PowerPC processor architecture.

Problem 13.30 Describe the salient features of Tilera's TILE64 system-on-a-chip, and the use of iMesh network to realize cache coherent NUMA architecture. Draw the schematic block diagram of a single tile.

Problem 13.31 A single processing tile in the TILE64 system does not involve very aggressive exploitation of ILP. Justify why this is the right choice in the context of the overall system design and the typical target applications.

Problem 13.32 Using a block diagram, describe in brief the salient features of the Sun UltraSparc T2 processor. Is the processor more suitable for intensive scientific and engineering computations, or for commercial servers and virtualization? Justify your answer in brief.

Problem 13.33 Describe in brief the salient features of the AMD Opteron processor. Comment briefly on whether such a processor should provide hardware support for multi-threading.

Problem 13.34 Compare and contrast the architectures of TILE64, Sun UltraSparc T2, and AMD Opteron processors.

Problem 13.35 Using a block diagram, describe in brief the salient features of Intel's Pentium IV processor. List the processor features designed for exploiting ILP. What are micro-operations? Why was it necessary to introduce that concept in the VLSI implementation of the processor?

Problem 13.36 Parallel programming language Chapel claims to support a global view of parallel programming. Explain this concept in brief, contrasting it with a fragmented view of parallel programming.

Problem 13.37 Explain in brief the concept of domain and subdomain in Chapel, giving an example of each. Show how a domain can be used in loop control.

Problem 13.38 In parallel programming language Chapel, describe in brief the functions of:

(a) `begin`, `cobegin`, `coforall` and `on` statements,

(b) readFE and writeEF synchronization operations, and

(c) Atomic operations.

Problem 13.39 Describe in brief the salient features of MPI, openMP, PThreads and PVM models for implementing parallel processing applications.

Problem 13.40 For a particular application, processing must be carried out using a three-stage software pipeline, as illustrated in Fig. 13.17. Note that this model of processing is justified if the ratio of computation to communication is high.

In Fig. 13.17, Process A reads a record from a database, and performs the first stage of processing on it. It then sends the record to Process B for the second stage of processing, and B then sends the record to Process C. After the third stage of processing, Process C writes the processed data to another database. Each process works only on one record at a time; in other words, multiple records are not buffered in any process, and there is no multi-threading within any process.

Design and implement this system using MPI.

Problem 13.41 How will the system design of Exercise 39 change if each process has a buffer which stores *N* records? What are the likely benefits of making this change?

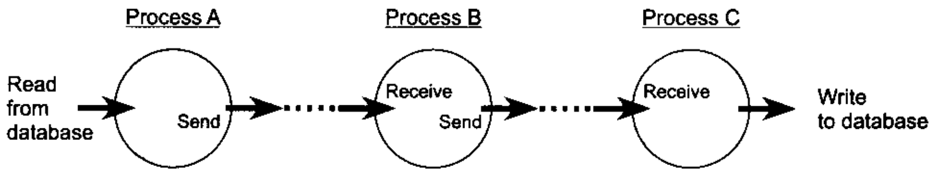


Fig. 13.17 Schematic of application for Exercise 39

Answers to Selected Exercises

Provided below are brief or partial answers to a few selected exercise problems. These answers are meant for readers to verify the correctness of their answers. Derivations or detailed computational steps in obtaining these answers are left for readers.

Exercise 1.1 Average CPI = 1.55. Effective processor performance = 258 MIPS. Execution time = 3.87 ms.

Exercise 1.4

(a) Average CPI = 2.24

(b) MIPS rate = 178.6

Exercise 1.8

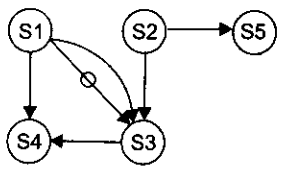
(a) Sequential execution time = 1664 CPU cycles.

(b) SIMD execution time = 26 machine cycles.

(c) Speedup factor = 64.

Exercise 2.5

(a)



(b) S_4 and S_5 need to use the same Store Unit in accessing the memory. Therefore they are potentially storage-dependent.

Exercise 2.11

(a)

Network	d	D	l	$(d \times D \times l)^{-1}$	Rank
Torus	6	6	192	1/6912	2
6-cube	6	6	192	1/6912	2
CCC	3	9	96	1/2592	1

(b)

Network	Mean Internode Distance	Rank
Torus	2.67	1
6-cube	2.67	1
CCC	3.67	2

Exercise 2.14

(a) A 4×4 switch has 256 legitimate input-output connections, 24 of which are permutations.

(b) A 64-input Omega network requires the use of 48 4×4 switches in 3 stages, with 16 switches per stage. Interstage connections are 4-way shuffles, 24^{48} permutations can be implemented in one pass through the network.

(c) The percentage of one-pass permutations equals $24^{48}/64! \approx 1.4 \times 10^{-23}$.

Exercise 3.2

(a) Effective speedup = 3. Vectorization ratio $\alpha = 0.75$.

(b) New speedup = 3.43 with vector/scalar speed ratio = 18.

(c) α must be improved to 0.8.

Exercise 3.3

(a) MIPS rate $= nx/[\alpha + n(1 - \alpha)]$;

(b) $\alpha = 0.96$

Exercise 3.7

(a) Sequential execution time = 1,051,628 cycles.

(b) The speedup = 16.28.

(c) Each processor is assigned 32 iterations balanced between the beginning and ending of the I-loop.

(d) The ideal speedup 32 is achieved.

Exercise 3.9

Machine	Arithmetic Mean Execution Time	Harmonic Mean Execution Rate	Rank
A	4.00 μs per instruction	0.25 MIPS	2
B	4.78 μs per instruction	0.21 MIPS	3
C	0.48 μs per instruction	2.10 MIPS	1

Exercise 4.11

(a) Average cost $c = (c_1s_1 + c_2s_2)/(s_1 + s_2).c \rightarrow c_2$ when $s_2 \gg s_1$ and $c_2s_2 \gg c_1s_1$.

(b) $t_a = ht_1 + (1 - h)t_2$.

(c) $E = \dfrac{1}{h + (1 - h)r}$.

(e) $h = 0.99$.

Exercise 4.15

(a) Hit ratio $h = 16/33$ for LRU policy.

(b) Hit ratio $h = 16/33$ for the circular FIFO policy.

(c) These two policies are equally effective for this particular page trace.

Exercise 4.17

(a) $t_{eff} = 0.95t_1 + 0.05t_2$.

(b) Total cost $c = c_1 s_1 + c_2 s_2$.

(c) s_2 cannot exceed 18.6 Mbytes, $t_2 = 420ns$.

Exercise 5.12

(a) $t_a = f_i (h_i c + (1 - h_i)(b + c) + (1 - f_i)(h_d) c + (1 - h_d)((b + c)(1 - f_{dir}) + (2b + c) f_{dir}))$.

(b) $t'_a = t_a + (1 - f_i) f_{inv}\, i$.

Exercise 5.13

(a) MIPS rate $= px(1 + mtx)$;

(b) $x = 583$ MIPS;

(c) Effective MIPS = 1524.

Exercise 5.17

(a) There are 20 program orders: abcdef, abdcef, abdecf, abdefc, adbcef, adbecf, adbefc, adebcf, adebfc, adefbc, dabcef, dabecf, dabefc, daebcf, daebfc, daefbc, deabcf, deabfc, deafbc, defabc.

(b) Possible output pattern: 0111, 1011, and 1111.

(c) Possible outputs are: 1001, 1011, 1101, 0110, 0111, 1110, and 1111.

Exercise 6.1 Under the favourable assumptions made, the time taken to initially fill the pipeline (i.e. 5 clock cycles) is a negligible fraction of the total execution time. Therefore the speedup, efficiency and throughput of the processor almost equal 5, 1 and 1000 MIPS, respectively.

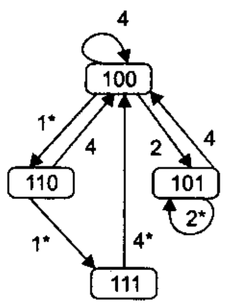
If we assume that the pipeline is flushed after every 100 instructions (on average), then 5 clock cycles are lost out of every 100, leading to a 5% loss in speedup, efficiency and throughput; in this case, the three answers are 4.75, 0.95 and 950, respectively.

Clearly the loss in speedup, efficiency and throughput will be greater if the pipeline is flushed more frequently, e.g. after every twenty instructions on average.

Exercise 6.9

(a) Forbidden latency is 3 with a collision vector (100).

(b) State transition diagram is shown below:



(c) Simple cycle: (2), (4), (1,4), (1,1,4), and (2,4);

(d) Optimal constant latency cycle: (2), MAL = 2.

(e) Throughput = 250 MIPS.

Exercise 6.15

(a) Speedup factor = 3.19;

(b) 62.5 MIPS for processor X and 199.2 MIPS for processor Y.

Exercise 6.17

(a) The four stages perform: Exponent subtract, Align, Fraction Add, and Normalize, respectively.

(b) 111 cycles to add 100 floating-point numbers.

Exercise 7.1

(a) Memory bandwidth = $mc/(c + m)\tau$ = 533 million words per second.

(b) Memory utilization = $cm/(c + m)$ = 5.33 requests per memory cycle.

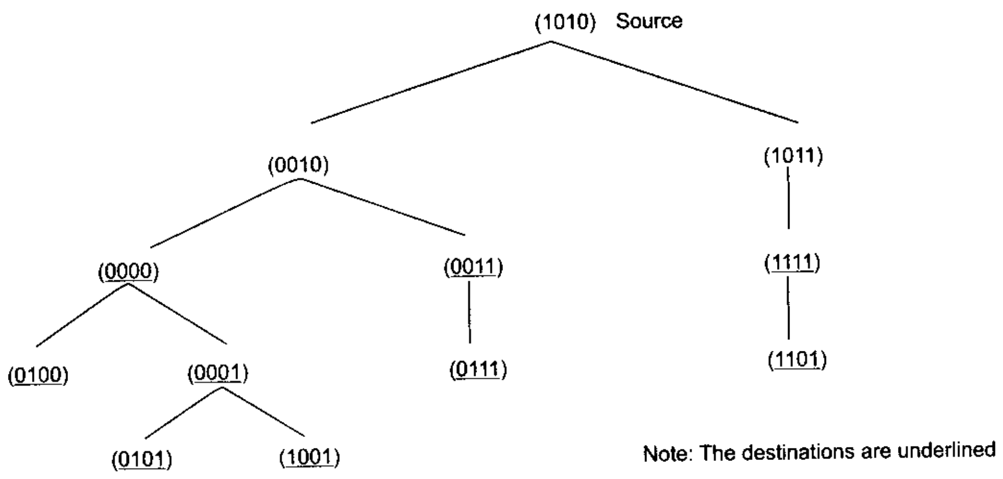
Exercise 7.4 A minimum of 21 time steps are needed to schedule the 24 code segments on the 4 processors, which are updating the 6 memory modules simultaneously without conflicts. The average memory bandwidth is thus equal to 70/21= 3.33 words per time step, where 70 accounts for 70 memory accesses by four processors in 21 time steps without conflicts.

Exercise 7.14

(a) (101101) → (101100) → (101110) → (101010) → (111010) → (011010)

(b) Use either a route with a minimum of 20 channels and distance 9 or another route with a minimum distance of 8 and 22 channels.

(c) The following tree shows multicast route on the hypercube:



Exercise 8.12

(a) $R_\alpha = 2000/(10 - 9\alpha)$, in Mflops;

(b) Vectorization ratio $\alpha = 26/27 = 0.963$;

(c) $R_v = 700$ Mflops.

Exercise 8.13

(a) Serial execution time = 190 time units.

(b) SIMD execution time = 35 time units.

Exercise 8.14

(a) C90 can execute 64 opreations per cycle of 4.2 ns, resulting a peak performance = $64/4.2 \times 10^{-9}$ = 15.2 Gflops.

(b) Similarly NEC has a peak performance of $64/2.9 \times 10^{-9}$ = 22 Gflops.

Exercise 9.1

(a) $E = 1/(1 + RL)$.

(b) $R' = (1 - h)R$, $E = 1/(1 + R'L) = 1/[1 + RL(1 - h)]$.

(c) While $N \geq N_d = \dfrac{L}{1/R' + C} + 1$, $E_{\text{sat}} = \dfrac{1}{1 + CR'} = \dfrac{1}{1 + (1 - h)CR}$.

While $N \leq N_d$, $E_{\text{lin}} = \dfrac{N/R'}{1/R' + C + L} = \dfrac{N}{1 + R'C + R'L} = \dfrac{N}{1 + (1 - h)(L + C)R}$.

(d) The mean internode distance $\bar{D} = (r + 4)/3$.

Thus $L = 2\bar{D}t_d + t_m = \dfrac{2(r + 4)}{3} t_d + t_m = \dfrac{2(\sqrt{p} + 4)}{3} t_d + t_m$.

$$E_{\text{sat}} = \frac{1/R'}{1/R' + C} = \frac{1}{1 + (1 - h)CR}$$

$$E_{\text{lin}} = \frac{N}{1 + (1 - h)R(L + C)} = \frac{N}{1 + (1 - h)R\left(\left[\dfrac{2(\sqrt{p} + 4)}{3} t_d + t_m\right] + C\right)}$$

Exercise 10.5

(a) A(5,8: *,*) declares A(5,8,1), A(5,9,1), A(5,10,1), A(5,8,2), A(5,9,2), A(5,10,2), A(5,8,3), A(5,9,3), A(5,10,3), A(5,8,4), A(5,9,4), A(5,10,4), A(5,8,5), A(5,9,5), A(5,10,5). B(3:*:3,5:8) corresponds to B(3,5), B(3,6), B(3,7), B(3,8), B(6,5), B(6,6), B(6,7), B(6,8), B(9,5), B(9,6), B(9,7), B(9,8). C(*,3,4) stands for C(1,3,4), C(2,3,4), C(3,3,4).

(b) Yes, no, no, and yes respectively for the four array assignments.

Exercise 10.7

(a) $S_1 \rightarrow S_2 \not\leftarrow S_3$

(b) S_1: A(1:N) = B(1:N)
S_3: E(1:N) = C(2:N+1)
S_2: C(1:N) = A(1:N) + B(1:N)

Exercise 10.12

(a) Vectorized code:

```
TEMP(1:N) = A(1:N)
A(2:N + 1) = TEMP(1:N) + 3.14159
```

(b) Parallelized code:

```
Doall I = 1, N
    If (A(I) .LE. 0.0) then
        S = S + B(I) * C(I)
        X = B(I)
    Endif
Enddo
```

Exercise 11.15

(a) Suppose the image is partitioned into p segments, each consisting of $s = m/p$ rows. Vector *histog* is shared among the processors. Therefore its update has to be performed in a critical section to avoid race conditions. Assume it is possible protect each element of vector *histog* by a separate semaphore. The following program performs parallel histogramming:

```
Var pixel(0 : $m - 1$,0 : $n - 1$);
Var histog(0 : $b - 1$): integer;
Var lock(0 : $b - 1$): [0,1];
histog(0 : $b - 1$) = 0;
lock(0 : $b - 1$) = 1;
for $k = 0$ until $p - 1$ 0 Doall
    for $i = k \times s$ until $(k + 1) \times s - 1$ do
        for $j = 0$ until $n - 1$ do
            P(lock(pixel($i, j$)));
            histog(pixel($i, j$)) = histog(pixel($i, j$)) + 1;
            V(lock(pixel($i, j$)));
        Enddo
    Enddo
Endall
```

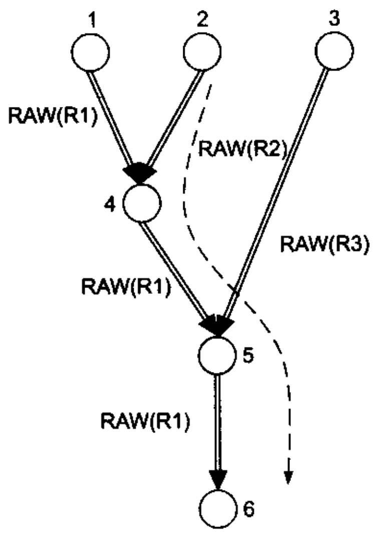
(b) The maximum speedup of the parallel program over the serial program is p, provided there is no conflict in accessing the *histog* vector and the overhead associated with synchronization is negligible. An alternative approach is to associate a local histog vector with each processor, which will obviate the use of critical sections. At the end of the algorithm, the values in local histog vectors are added to obtain the final result.

Exercise 12.9

Note: The aim in Chapter 12 has been to understand instruction level parallelism without reference to a specific processor design. The stipulation of *counting from the last clock cycle of instruction* 1 has been added to these exercises so that the instruction sequences can be analyzed without reference to a specific processor pipeline design. Thus the answers we derive do not include the initial pipeline fill time, and we count only the additional clock cycles needed to complete each instruction.

```
Sequence 1:  1 LOAD     mem-a, R1
             2 LOAD     mem-b, R2
             3 LOAD     mem-c, R3
             4 FADD     R2, R1, R1
             5 FSUB     R3, R1, R1
             6 STORE    mem-a, R1
```

The directed graph of dependences is shown below:



Exercise 12.10 Here we assume that the processor has no provision for register renaming and operand forwarding, and that all memory references are satisfied from L1 cache.

In the absence of operand forwarding, every RAW hazard causes (at least) one lost clock cycle, as the operand value is first written into the register and then, in the next cycle, brought to the functional unit or load/store unit.

Instruction 3 can be executed in parallel with instruction 4 (FADD).

We therefore add the additional clock cycles required for the rest of the instructions (other than instruction 3), and add one cycle penalty for every RAW dependence occurring along the dotted path. Thus the number of cycles needed (from the last cycle of instruction 1) equals:

$$1 + 1 + 2 + 2 + 1 + 3*1 = 10$$

Exercise 12.11 Effect of register renaming:

No WAR or WAW dependences are affecting the computation, and therefore register renaming will not yield any additional parallelism in this particular instance.

Exercise 12.12 and 12.13 Effect of operand forwarding:

With operand forwarding, the FPU or memory store unit receives its required operand value in the same cycle in which it is written in the register. Therefore the three cycles lost due to RAW dependences are saved, and the answer is:

$$1 + 1 + 2 + 2 + 1 = 7$$

Exercise 12.14 Effect of L1 cache miss, requiring L2 cache access of 5 clock cycles:
L1 cache miss on instruction 1 or 2 will cost $5 - 1 = 4$ additional clock cycles each.
L1 cache miss on instruction 3 will cost $5 - 2 = 3$ additional clock cycles, since 2 cycles out of five are in parallel with FADD of instruction 4.

Exercise 12.15 Outline of Tomasulo's algorithm: For every possible source of an operand within the processor, assume a tag value. For example, assume a tag value of TFPU for the output of FPU, and TLOAD for the output of memory load unit.

For every RAW dependence, if the operand value is not available, the algorithm requires the source tag value to be written into the destination tag register. When the operand value is available on the CDB (along with the right tag value), it is copied into every destination register where it is required, i.e. where the source and destination tag values match.

However, in this case the memory load unit requires special care, since its successive outputs from instructions 1, 2 and 3 go (respectively) to the two inputs of FPU for instruction 4, and then again an input of FPU for instruction 5. One way to handle this would be to assign multiple tag values to the output of memory load unit, and use different values for the three load operations of instructions 1, 2 and 3.

Exercise 13.39 and 13.40 Note that the first two processes make up one producer-consumer pair, and the last two processes make up another producer-consumer pair. Process B consumes the records produced by process A, and process C consumes the records produced by process B. The solutions require two applications of the standard producer-consumer algorithm.

Bibliography

[Accetta86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proc. Summer 1986 USENIX Conf.*, pp. 93–113, Atlanta, GA, June 1986.

[ACM91] ACM, *Resources in Parallel and Concurrent Systems with an Introduction by Charles Seitz,* ACM Press, New York, 1991.

[Acosta86] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Trans. Computers*, pp. 815-825, Sept. 1986.

[Adam74] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Commun. ACM,* 17(12):685–690, 1974.

[Adve90] S. V. Adve and M. D. Hill, "Weak Ordering: A New Definition," *Proc. 17th Annu. Int. Symp. Computer Arch.*, 1990.

[Adve91] S. V. Adve, V. S. Adve, M. D. Hill, and M. Vernon, "Comparison of Hardware and Software Cache and Coherence Schemes," *Proc. 18th Annu. Int. Symp. Computer Arch.*, pp. 298–308, 1991.

[Agarwal88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Annu. Int. Symp. Computer Arch.*, 1988.

[Agarwal90] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," *Proc. 17th Annu. Int. Symp. Computer Arch.*, pp. 104–114, 1990.

[Agarwal91] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. A. Yeung, "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," *Proc. Workshop Multithreaded Computers, Supercomputing 91*, 1991.

[Agarwal92a] A. Agarwal, "Performance Tradeoffs in Multithread Processors," *IEEE Trans. Parallel Distri. Systems,* 3(5):525–539,1992.

[Agarwal92b] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kobiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Mussbaum, "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," in Dubois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors,* Kluwer Academic Publishers, Boston, MA, 1992.

[Agha86] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Press, Cambridge, MA, 1986.

[Agha90] G. Agha, "Concurrent Object-Oriented Programming," *Commun. ACM,* 33(9):125–141, Sept. 1990.

[Aho74] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1974.

[Ahuja86] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer,* 19(8):16–34, 1986.

[Allan85] S. J. Allan and R. Oldehoeft, "HEP SISAL: Parallel Functional Programming," in Kowalik (ed.), *Parallel MIMD Computation: HEP Supercomputers and Applications,* MIT Press, Cambridge, MA, 1985.

[Allen84] J. R. Allen and K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Fortran," in Hwang (ed.), *Supercomputers: Design and Applications,* IEEE Computer Society Press, Los Alamitos, CA, 1984.

[Allen87] R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," *ACM Trans. Prog. Lang. and Systems*, pp. 491–542, Oct. 1987.

[Alliant89] Alliant, *Alliant Product Summary*, Alliant Computer Systems Corporation, Littleton, MA, 1989.

[Almasi89] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, Redwood, CA, 1989.

[Alverson90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," *Proc. ACM Int. Conf. Supercomputing*, pp. 1–6, Amsterdam, The Netherlands, June 1990.

[Amdahl67] G. M. Amdahl, "Validity of Single-Processor Approach to Achieving Large-Scale Computing Capability," *Proc. AFIPS Conf.*, pp. 483–485, Reston, VA., 1967.

[Anaratone86] M. Anaratone, E. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, K. Sarocky, and J. A. Webb, "Warp Architecture and Implementation," *Proc. 13th Annu. Int. Symp. Computer Arch.*, pp. 346–356, Tokyo, June 1986.

[Anderson67] D. W. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powere, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM Systems Journal*, pp, 34–53, Feb. 1967.

[Andrews91] G. R. Andrews, *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, Redwood, CA, 1991.

[Archibald86] J. Archibald and J. L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems*, 4(4):273–298, Nov. 1986.

[Arden81] B. W. Arden and H. Lee, "Analysis of Chordal Ring Network," *IEEE Trans. Computers*, 30(4):291–295, 1981.

[Arvind83] Arvind and R. A. Iannucci, "A Critique of Multiprocessing von Neumann Style," *Proc. 10th Symp. Computer Arch.*, Stockholm, Sweden, 1983.

[Arvind84] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas, "The Tagged Token Dataflow Architecture," Technical Report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1984.

[Arvind87] Arvind and R. A. Iannucci, "Two Fundamental Issues in Multiprocessing," *Proc. Parallel Processing on Science and Engineering*, 1987.

[Arvind90] Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Trans. Computers*, 39(3):300–318, 1990.

[Arvind91] Arvind, L. Bic, and T. Ungerer, "Evolution of Dataflow Computers," in Gaudiot and Bic (eds.), *Advanced Topics in Dataflow Computing*, pp. 3–34, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Athas88] W. C. Athas and C. L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, 21(8):9–24, Aug. 1988.

[Axelrod86] T. S. Axelrod, "Effects of Synchronization Barriers on Multiprocessor Performance," *Parallel Computing*, 3(2):129–140, May 1986.

[Babb88] R. G. Babb, *Programming Parallel Processors*, Addison-Wesley, Reading, MA, 1988.

[Bach84] M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Operating Systems," *AT&T Bell Lab. Tech. Journal*, 63(8), Oct. 1984.

[Bach86] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Baer80] J. L. Baer, *Computer Systems Architecture*, Computer Science Press, Rockville, MD, 1980.

[Banerjee79] U. Banerjee, *Speedup of Ordinary Programs*, Ph.D. thesis, University of Illinois, 1979.

[Banerjee88] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Press, Boston, MA, 1988.

[Barnes68] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer," *IEEE Trans. Computers*, pp. 746–757, Aug. 1968.

[Batcher76] K. Batcher, "The Flip Network in STARAN," *Proc. Int. Conf. Parallel Processing*, pp. 65–71, 1976.

[Batcher80] K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. Computers*, pp. 836–840, Sept. 1980.

[BBN89]. BBN Advanced Computers Inc., Cambridge, MA., *TC2000 Technical Product Summary*, Nov. 1989.

[Beetem85] J. Beetem, M. Denneau, and D. Weingarten, "The GF11 Supercomputer," *Proc. 12th Annu. Int. Symp. Computer Arch.*, pp. 363–376, Boston, MA, May 1985.

[Bell92] G. Bell, "Ultracomputer: A Teraflop Before Its Time," *Commun. ACM*, 35(8):27–47, 1992.

[Ben-Ari90] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Bernstein66] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Trans. Computers*, pp. 746–757, Oct. 1966.

[Berntsen90] J. Berntsen, "Communication-Efficient Matrix Multiplication on Hypercubes," *Parallel Computing*, pp. 335–342, 1990.

[Bhuyan83] L. N. Bhuyan and D. P. Agrawal, "Design and Performance of Generalized Interconnection Networks," *IEEE Trans. Computers*, pp. 1081–1090, Dec. 1983.

[Bisiani88] R. Bisiani and M. Ravishankar, "Plus: A Distributed Shared-Memory System," *Proc. 17th Annu. Int. Symp. Computer Arch.*, pp. 115–124, 1988.

[Bitar86] P. Bitar and A. M. Despain, "Multiprocessor Cache Synchronization: Issues, Innovations, and Evolution," *Proc. 13th Annu. Int. Symp. Computer Arch.*, 1986.

[Bitar91] P. Bitar, "MIMD Synchronization and Coherence," Technical Report UCB/CSD 90/605, University of California, Berkeley, May 1991.

[Bitar92] P. Bitar, "The Weakest Memory-Access Order," *J. Para. Distri. Computing*, 15:305–331, 1992.

[Black90] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer*, 23(5):35–42, May 1990.

[Blelloch90] G. E. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press, Cambridge, MA, 1990.

[Blevins90] D. W. Blevins, E. W. Davis, R. Heaton, and J. H. Reif, "BLITZEN: A Highly Integrated Massively Parallel Machine," *J. Para. Distri. Computing*, pp. 150–160, 1990.

[Boothe92] B. Boothe and A. Ranade, "Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessor," *Proc. 19th Annu. Int. Symp. Computer Arch.*, Australia, May 1992.

[Borkar90] S. Borkar, R. Cohn, G. Fox, T. Gross, H. Hung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb, "Supporting Systolic and Memory Communication in iWARP," *Proc. 17th Annu. Int. Symp. Computer Arch.*, pp. 70–81, May 1990.

[Brainerd90] W. S. Brainerd, C. H. Golberg, and J. C. Adams, *Programmer's Guide to Fortran 90*, McGraw-Hill, New York, 1990.

[Brawer89] S. Brawer, *Introduction to Parallel Programming*, Academic Press, New York, 1989.

[Briggs82] F. A. Briggs, K. S. Fu, K. Hwang, and B. W. Wah, "PUMPS Architecture for Pattern Analysis and Image Database Management," *IEEE Trans. Computers*, pp. 969–982, Oct. 1982.

[Brinch Hansen75] P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE 'Trans. Software Engineering*, SE-1(2):199–206, June 1975.

[Brunner90] R. A. Brunner, D. P Bhandarkar, F. X. McKeen, B. Patel, W. J. Rogers, and G. L. Yoder, "Vector Processing on the VAX 9000 System," *Digital Technical Journal*, 2(4):61–79, 1990.

[Burkhardt92] H. Burkhardt, *Technical Summary of KSR-1*, Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA 02154, 1992.

[Butler91] M. Butler, T. Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single-Instruction-Stream Parallelism is Greater Than Two," *Proc. 18th Annu. Int. Symp. Computer Arch.*, pp. 276–286, 1991.

[Buzbee83] B. L. Buzbee et al., "Supercomputing Value and Trends," unpublished slide presentation, Los Alamos National Laboratory, NM, July 1983.

[Callahan85] D. Callahan, "Task Granularity Studies on a Many-Processor CRAY X-MP," *Parallel Computing*, pp. 109–118, June 1985.

[Callahan88] D. Callahan, K. Cooper, R. Hood.K, Kennedy, and L. Torczon, "ParaScope: A Parallel Programming Environment," *Int. J. Supercomputer Appl.*, 2(4), 1988.

[Caswell90] D. Caswell and D. Black, "Implementing a Mach Debugger for Multithread Applications," *Proc. Winter 1990 USENIX Conf.*, Washington, DC, Jan. 1990.

[CDC80] CDC, *Cyber 200/Model 205 Technical Description*, Control Data Corporation, Nov. 1980.

[CDC90] CDC, "Introduction to Cyber 2000 Architecture," Technical Report 60000457, Control Data Corporation, St. Paul, MN, 1990.

[Cekleov90] M. Cekleov, M. Dubois, J. C. Wang, and F. A. Briggs, "Virtual-Address Cache in Multiprocessors," in Dubois and Thakkar (eds.), *Cache and Interconnect Architectures in Multiprocessors*, Kluwer Academic Press, Boston, MA, 1990.

[Censier78] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, C-27(12):1112–1118, Dec. 1978.

[Chaiken90] D. Chaiken, C. Fields, K. Kwihara, and A. Agrawal, "Directory-Based Cache Coherence in Large-Scale Multiprocessor," *IEEE Computer*, 23(6):49–59, 1990.

[Chang88] A. Chang and M. F. Mergen, "801 Storage: Architecture and Programming," *ACM Trans. Computer Systems*, 6(1):28–50, 1988.

[Chang90] L. C. Chang and B. T. Smith, "Classification and Evaluation of Parallel Programming Tools," Technical Report CS 90-22, University of New Mexico, Albuquerque, NM 87131, 1990.

[Chang91] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W. M Hwu, "Computing Static and Dynamic Code Scheduling for Multiple-Instruction-Issue Processors," *Proc. 24th Int. Symp. Microarch.*, 1991.

[Cheng89] H. Cheng, "Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X/MP," *IEEE Computer*, 22(9):31–44, 1989.

[Cheng91] D. Y. Cheng, "A Survey of Parallel Programming Tools," Technical Report RND-91-005, NASA Ames Research Center, Moffett Field, CA, 1991.

[Chin84] C. Y. Chin and K. Hwang, "Packet Switching Networks for Multiprocessors and Dataflow Computers," *IEEE Trans. Computers*, pp. 991–1003, Nov. 1984.

[Chow74] C. K. Chow, "On Optimization for Storage Hierarchies," *IBM J. Res. and Develop.*, pp. 194–203, 1974.

[Christy90] P. Christy, "Software to Support Massively Parallel Computing on the MasPar MP-1," *Digest of Papers Spring Compeon*, San Francisco, CA, Feb. 1990.

[Clark83] K. L. Clark and S. Gregory, "Parlog: A Parallel Logic Programming Language," Technical Report DOC 83-5, Dept. of Computing, Imperial College, London, May 1983.

[Clark86] R. S. Clark and T. L. Wilson, "Vector System Performance of the IBM 3090," *IBM Systems Journal*, 25(1), 1986.

[Cocke90] J. Cocke and V. Markstein, "The Evolution of RISC Technology at IBM," *IBM J. Res. and Deoelop.*, 34(1): 4–11, 1990.

[Convex90] Convex, *Fortran Optimization Guide*, Convex Computer Corporation, Richardson, TX,1990.

[Cormen90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[Cragon89] H. G. Cragon and W. J. Watson, "The TI Advanced Scientific Computer," *IEEE Computer*, 22(1):55–64, 1989.

[Cragon92a] H. G. Cragon, *Branch Strategy Taxonomy and Performance Models*, IEEE Computer Society Press, Los Alamitos, CA, 1992.

[Cragon92b] H. G. Cragon, "Memory Systems and Pipeline Processors," Class notes, Department of ECE, University of Texas, Austin, 1992.

[Crawford90] J. H. Crawford, "The i486 CPU: Executing Instructions in One Clock Cycle," *IEEE Micro*, 10(1):27–36, Feb. 1990.

[Cray77] Cray, *CRA Y-1 Computer System Hardware Reference Manual*, Cray Research Institute, 1977.

[Cray89] Cray, *The Cray Y/MP Functional Description Manual*, Cray Research Inc., Eagan, MN,1989.

[Cray91] Cray, *The Cray Y/MP C-90 Supercomputer System*, Cray Research Inc., Eagan, MN,1991.

[Cray92] Cray, *Cray/MPP Announcement*, Cray Research, Inc., Eagan, MN, 1992.

[CSRD91] CSRD, "Perfect Club Benchmark Evaluation Package," Technical report, Center for Supercomputer Research and Development, University of Illinois, Urbana, 1991.

[Cybenko92] G. Cybenko and D. J. Kuck, "Revolution or Evolution," *IEEE Spectrum*, 29(9):39–41, 1992.

[Dally86] W. J. Dally and C. L. Seitz, "The Torus Routing Chip," *Journal of Distributed Computing*, 1(3):187–196, 1986.

[Dally87a] W. J. Dally et al., "Architecture of a Message-Driven Processor," *Proc. 14th Annu. Int. Symp. Computer Arch.*, pp. 189–205, IEEE CS Press, June 1987.

[Dally87b] W. J. Dally and C. L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Network," *IEEE Trans. Computers*, C-36(5):547–553, May 1987.

[Dally87c] W. J. Dally and P. Song, "Design of a Self-Timed VLSI Multicomputer Communication Controller," *Proc. Int. Conf. Computer Design*, pp. 230–234, IEEE CS Press, Oct. 1987.

[Dally90a] W. J. Dally, "Network and Processor Architecture for Message-Driven Computers," in Suaya and Birtwistle (eds.), *VLSI and Parallel Computation*, Chapter 3, Morgan Kaurmann, San Mateo, CA, 1990.

[Dally90b] W. J. Dally, "Performance Analysis of k-ary n-Cube Interconnection Networks," *IEEE Trans. Computers*, 39(6):775–785, 1990.

[Dally90c] W. J. Dally, "Virtual Channel Flow Control," *Proc. 17th Annu. Int. Symp. Computer Arch.*, pp. 60–68, May 1990.

[Dally92] W. J. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler, "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," *IEEE Micro*, 12(2):23–39, Apr. 1992.

[Davidson71] E. S. Davidson, "The Design and Control of Pipelined Function Generators," *Proc. Int. IEEE Conf. System Networks and Computers*, pp. 19–21, 1971.

[Davidson75] E. S. Davidson, D. P. Thomas, L. E. Shar, and J. H. Patel, "Effective Control for Pipelined Computers," *Proc. COMPCON*, pp. 181–184, 1975.

[DEC86] DEC, "The Whetstone Performance," Technical report, Digital Equipment Corporation, Bedford, MA, 1986.

[DEC92] DEC, "Alpha Architecture Handbook," Technical report, Digital Equipment Corporation, Boxboro, MA, 1992.

[DeCegamma89] A. L. DeCegamma, *The Technology of Parallel Processing: Architectures and VLSI Hardware*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Dekel81] E. Dekel, D. Nassimi, and S. Sahni, "Parallel Matrix and Graph Algorithms," *SIAM J. Computing*, pp. 657–673, 1981.

[Denning68] P. J. Denning, "Working Set Model for Program Behavior," *Commun. ACM,* 11(6):323–333, 1968.

[Dennis80] J. B. Dennis, "Data Flow Supercomputers," *IEEE Computer,* pp. 48–56, Nov. 1980.

[Dennis91] J. Dennis, "The Evolution of "Static" Dataflow Architecture," in Gaudiot and Bic (eds.), *Advanced Topics in Dataflow Computing,* pp. 35–91, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Diefendorff92] K. Diefendorff and M. Allen, "Organization of the Motorola 88110 Superscalar RISC Microprocessor," *IEEE Micro,* 12(2):40–63, Apr. 1992.

[Dijkstra68] E. W. Dijkstra, "Cooperating Sequential Processes," in Genuys (ed.), *Programming Languages,* Academic Press, New York, 1968.

[Dinning89] A. Dinning, "A Survey of Synchronization Methods for Parallel Computers," *IEEE Computer,* 22(7), 1989.

[Dongarra86] J. J. Dongarra and D. C. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs," Technical Memo 86, Argonne National Laboratory, 1986.

[Dongarra89] J. Dongarra and A. Hinds, "Comparison of the Cray X/MP-4, Fujitsu VP-200, and Hitachi S-810/20," in Hwang and DeGroot (eds.), *Parallel Processing for Supercomputing and Artificial Intelligence,* pp. 289–323, McGraw-Hill, New York, 1989.

[Dongarra92] J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software," Technical report, Computer Science Department, University of Tennessee, Knoxville, TN, 1992.

[Dubois86] M. Dubois, C. Scheurich, and F. A. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. 13th Annu. Int. Symp. Computer Arch.,* pp. 434–442, 1986.

[Dubois88] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence and Event Ordering in Multiprocessors," *IEEE Computer,* 21(2), 1988.

[Dubois90a] M. Dubois and F. A. Briggs, "Tutorial Notes on Shared-Memory Architectures for Multiprocessors," *Proc. 17th Symp. Computer Arch.,* Seattle, WA, 1990.

[Dubois90b] M. Dubois and S. Thakkar (eds.), *Cache and Interconnect Architectures in Multiprocessors,* Kluwer Academic Publishers, Boston, MA, 1990.

[Dubois92a] M. Dubois, "Delayed Consistency," in Dubois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors,* Kluwer Academic Publishers, Boston, MA, 1992.

[Dubois92b] M. Dubois and S. Thakkar (eds.), *Scalable Shared-Memory Multiprocessors,* Kluwer Academic Publishers, Boston, MA, 1992.

[Eager89] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Trans. Computers,* 38(3):408–423, Mar. 1989.

[Edenfield90] R. Edenfield, M. Gallup, W. Ledbetter, R. McGarity, E. Quintana, and R. Reininger, "The 68040 Processor: Part 1. Design and Implementation," *IEEE Micro,* 10(1):66–78, Feb. 1990.

[Emma87] P. G. Emma and E. S. Davidson, "Characterization of Branch and Data Dependences in Programs for Evaluating Pipeline Performance," *IEEE Trans. Computers,* 36:859–875, 1987.

[Encore87] Encore, *Multimax Technical Summary,* Encore Computer Corporation, Ft. Lauderdale, FL, Mar. 1987.

[Enslow74) P. H. Enslow (ed.), *Multiprocessors and Parallel Processing,* Wiley, New York, 1974.

[Felten85] E. Felten, S. Karlin, and S. W. Otto, "The Traveling Salesman on a Hypercube MIMD Computer," *Proc. Int. Conf. Parallel Processing,* St. Charles, IL, Aug. 1985.

[Feng81] T. Y. Feng, "A Survey of Interconnection Networks," *IEEE Computer,* 14(12):12–27, 1981.

[Ferrante87] M. W. Ferrante, "Cyberplus and Map V Interprocessor Communications for Parallel and Array Processor Systems," in Karplus (ed.), *Multiprocessors and Array Processors,* Simulation Councils, Inc., San Diego, CA, 1987.

[Fisher81] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers,* 30(7):478–490, 1981.

[Fisher83] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proc. 10th Symp. Computer Arch.*, pp. 140–150, ACM Press, New York, 1983.

[Flynn72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Computers,* 21(9):948–960, 1972.

[Fortune78] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," *Proc. ACM Symp. Theory of Computing,* pp. 114–118, 1978.

[Fox87] G. C. Fox, S. W. Otto, and A. J. Hey, "Matrix Algorithms on Hypercube (I): Matrix Multiplication," *Parallel Computing,* pp. 17–31, 1987.

[Fujitsu90] Fujitsu, *VP2000 Series Supercomputers,* Fujitsu Ltd., Japan, 1990.

[Fujitsu92] Fujitsu, *VPP500 Vector Parallel Processor,* Fujitsu America, Inc., San Jose, CA, 1992.

[GaJski82] D. D. Gajski, D. Padua, D. J. Kuck, and R. H. Kuhn, "A Second Opinion on Dataflow Machines and Languages," *IEEE Computer,* 15(2), 1982.

[Gajski85] D. D. Gajski and J. K. Peir, "Essential Issues in Multiprocessor Systems," *IEEE Computer,* 18(6), 1985.

[Gaudiot91] J.-L. Gaudiot and L. Bic, *Advanced Topics in Dataflow Computing,* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Gelernter85a] D. Gelernter, "Generative Communication in Linda," *ACM Trans. Prog. Lang. and Systems,* 7(1):80–112, Jan. 1985.

[Gelernter85b] D. Gelernter, N. Carriero, S. Chandran, and S. Chang, "Parallel Programming in Linda," *Proc. Int. Conf. Parallel Processing,* pp. 255-263, 1985.

[Gelernter90] D. Gelernter, A. Nicolau, and D. Padua, *Languages and Compilers for Parallel Computing,* MIT Press, Cambridge, MA, 1990.

[Gharachorloo90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Annu. Int. Symp. Computer Arch.*, June 1990.

[Gharachorloo91] G. M. Gharachorloo and K. R. Traub, "Multithreading: A Revisionist View of Dataflow Architecture," *Proc. 18th Annu. Int. Symp. Computer Arch.*, May 1991.

[Gharachorloo92a] K. Gharachorloo, S. Adve, A. Gupta, J. L. Hennessy, and M. Hill, "Programming for Different Memory Consistency Models," *J. Para. Disiri. Computing,* Aug. 1992.

[Gharachorlo092b] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors," *Proc. 19th Annu. Int. Symp. Computer Arch.*, Gold Coast, Australia, May 1992.

[Gharachorlo092c] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proc. Fourth Int. Conf. Arch. Support for Prog. Lang. and OS,* 1992.

[Gjessing92] S. Gjessing, G. B. Gustavson, J. R. James, and E. H. Kristiansen, "The SCI Cache Coherence Protocol," in Dubois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors,* Kluwer Academic Publishers, Boston, MA, 1992.

[Glass92] C. J. Glass and 1. M. Ni, "The TURN Model for Adaptive Routing," *Proc. 19th Annu. Int. Symp. Computer Arch.*, 1992.

[Goble81] G. H. Goble and M. H. Marsh, "A Dual-Processor UNIX VAX 11/780," Technical report, Dept. of Electrical Engineering, Purdue University, West Lafayette, IN, Sept. 1981.

[Goff91] G. Goff, K. Kennedy, and C. W. Tseng, "Practical Dependence Testing," *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implementation*, 1991.

[Goodman83] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Symp. Computer Arch.*, pp. 124–131, June 1983.

[Goodman88] J. R. Goodman and P. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," *Proc. 15th Annu. Int. Symp. Computer Arch.*, pp. 422–431, 1988.

[Goodman89] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proc. Third Int. Conf. Arch. Support for Prog. Lang. and OS*, pp. 64–73, 1989.

[Goodman90] J. R. Goodman, "Cache Consistency and Sequential Consistency," Technical Report 61, IEEE SCI Committee, 1990.

[Goor89] A. J. van de Goor, *Computer Architecture and Design*, Addison-Wesley, Reading, MA,1989.

[Gottlieb83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer–Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers*, C-32(2):175–189, February 1983.

[Goyal84] A. Goyal and T. Agerwala, "Performance Analysis of Future Shared-Storage Systems," *IBM J. Res. and Develop.* pp. 95–98, Jan. 1984.

[Graham92] S. Graham, J. L. Hennessy, and J. D. Ullman, "Course on Code Optimization and Code Generation," in *Tutorial on Code Optimization and Generation*, Western Institute of Computer Science, Stanford University, Aug. 1992.

[Graunke90] G. Graunke and S. Thakkar, "Synchronization Algorithms for Shared-Memory Multiprocessors," *IEEE Computer*, 23(6):60–69, 1990.

[Greenberg89] R. I. Greenberg and C. E. Leiserson, "Randomized Routing on Fat Trees," *Advances in Computing Research*, 7:345–374, 1989.

[Gross83] T. R. Gross, "Code Optimization Techniques for Pipelined Architectures," *Proc. IEEE Computer Society Spring Int. Conf.*, pp. 278–285, 1983.

[Gupta90] A. Gupta, W. D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache coherence Schemes," *Proc. Int. Conf. Parallel Processing*, pp. 312–321, 1990.

[Gupta91] A. Gupta, J. L. Hennessy, K. Gharachorloo, T. Mowry, and W. D. Weber, "Computative Evaluation of Latency Reducing and Tolerating Techniques," *Proc. 18th Annu. Int. Symp. Computer Arch.*, pp. 254–263, Toronto, May 1991.

[Gupta92] A. Gupta and V. Kumar, "Scalability of Parallel Algorithms for Matrix Multiplication," Technical report, University of Minnesota, 1992.

[Gupta93] A. Gupta and V. Kumar, "The Scalability of FFT on Parallel Computers," *IEEE Trans. Parallel Distri. Systems*, to appear in 1993.

[Gurd85] J. R. Gurd, C. Kirkham, and J. Watson, "The Manchester Prototype Dataflow Computer," *Commun. ACM*, 28(1):36–45, 1985.

[Gustafson86] J. I. Gustafson, S. Hawkinson, and K. Scott, "The Architecture of a Homogeneous Vector Supercomputer," *Proc. Int. Conf. Parallel Processing*, pp. 649–652, 1986.

[Gustafson88] J.L. Gustafson, "Reevaluating Amdahl's Law," *Commun. ACM*, 31(5):532–533, May 1988.

[Gustafson91] J. Gustafson, D. Rover, S. Elbert, and M. Carter, "The Design of a Scalable, Fixed-Time Computer Benchmark," *J. Para. Distri. Computing*, 11, Aug. 1991.

[Gustavson86] D. B. Gustavson, "Introduction to the Fastbus," *Microprocessors and Microsystem*, 10(2):77–85, 1986.

[Hagersten92] E. Hagersten, A. Landin, and S. Haridi, "Multiprocessor Consistency and Synchronization Through Transient Cache States," in Dubois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors,* Kluwer Academic Publishers, Boston, MA, 1992.

[Halstead85] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Prog. Lang. and Systems*, 7(4):501–538, Oct. 1985 .

[Harrison90] W. Harrison, B. Kramer, W. Rudd, S. Shatz, C. Chang, Z. Segall, D. Clemmer, J. Williamson, B. Peek, B. Appelbe, K. Smith, and A. Kolawa, "Tools for Multiple-CPU Environments," *IEEE Software,* 7(3):45–51, May 1990.

[Hayes86] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "Architecture of a Hypercube Supercomputer," *Proc. Int. Conf. Parallel Processing,* pp. 653–660, 1986.

[Heath87] M. T. Heath, "Hypercube Applications at Oak Ridge National Laboratory," Heath (ed.), *Hypercube Multiprocessors,* SIAM, Philadelphia, 1987.

[Hellerman67] H. Hellerman, *Digital Computer System Principles,* McGraw-Hill, New York, 1967.

[Hennessy90] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann, San Mateo, CA, 1990.

[Hennessy92] J. L. Hennessy, "Introduction to a Tutorial on Code Optimization and Generation," WICS, Stanford University, 1992.

[Hertzberger84] L. O. Hertzberger, "The Architecture of the Fifth-Generation Inference Computers," *Future Generation Computer Systems,* 1(1):19–21, July 1984.

[Hill92] M. D. Hill, "What Is Scalability?," in Dubois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors,* Kluwer Academic Press, Boston, MA, 1992.

[Hillis86] W. D. Hillis and G. L. Steele, "Data Parallel Algorithms," *Commun. ACM,* 29(12):1170–1183, 1986.

[Hiraki87] K. Hiraki, K. Nishida, S. Sekiguchi, T. Shimada, and T. Yiba, "The SIGMA-l Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems," *J. Info. Processing,* 10(4):219–226, 1987.

[Hirata92] H. Hirata, K. Kimura, S. Nagamine, Mochizuki, A. Nishimura, and Y. Nakase, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," *Proc. 19th Annu. Int. Symp. Computer Arch.,* 1992.

[Hoare74] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Commun. ACM,* 17(10):549–557, Oct. 1974.

[Holt78] R. C. Holt, G. S. Graham, E. D. Lazowska, and M. A. Scott, *Structured Concurrent Programming with Operating Systems Applications,* Addison-Wesley, Reading, MA, 1978.

[Homewood87] M. Homewood, D. May, D. Shepherd, and R. Shepherd, "THE IMS T800 Transputer," *IEEE Micro,* 7(5):10–26, Oct. 1987.

[Hord90] R. M. Hord, *Parallel Supercomputing in SIMD Architectures,* CRC Press, Boca Raton, FL, 1990.

[Horwat89] W. Horwat, "Concurrent Smalltalk on the Message-Driven Processor," Master's thesis, Laboratory for Computer Science, MIT, 1989.

[Hotchips91] Hotchips, *Proc. Hot Chips III Symp. on High-Performance Chips,* Stanford University, Palo Alto, CA, 1991.

[Hwang77] K. Hwang and S. B. Yao, "Optimal Batched Searching of Tree-Structured Files in Multiprocessor System," *J. ACM,* pp. 441–454, July 1977.

[Hwang78] K. Hwang, *Computer Arithmetic: Principles, Architecture and Design,* Wiley, New York, 1978.

[Hwang82a] K. Hwang and Y. H. Cheng, "Partitioned Matrix Algorithms for VLSI Arithmetic Systems," *IEEE Trans. Computers,* pp. 1215–1224, Dec. 1982.

[Hwang82b] K. Hwang, W. Croft, G. H. Goble, B. W. Wah, F. A. Briggs, W. Simmons, and C. L. Coates, "A UNIX-Based Local Area Network with Load Balancing," *IEEE Computer,* 15(4):55–66, 1982.

[Hwang84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, New York, 1984.

[Hwang87a] K. Hwang, "Advanced Parallel Processing with Supercomputer Architectures," *Proc. IEEE,* vol. 75, Oct. 1987.

[Hwang87b] K. Hwang and J. Ghosh, "Hypernet: A Communication-Efficient Architecture for Constructing Massively Parallel Computers," *IEEE Trans. Computers,* 36:1450–1466, Dec. 1987.

[Hwang87c] K. Hwang, J. Ghosh, and R. Chowkwanyun, "Computer Architectures for AI Processing," *IEEE Computer,* pp. 19–29, Jan. 1987.

[Hwang88] K. Hwang and Z. Xu, "Multipipeline Networking for Compound Vector Processing," *IEEE Trans. Computers,* 37(1):33–47, 1988.

[Hwang89a] K. Hwang and D. DeGroot (eds.), *Parallel Processing for Supercomputers and Artificial Intelligence,* McGraw-Hill, New York, 1989.

[Hwang89b] K. Hwang, P. S. Tseng, and D. Kim, "An Orthogonal Multiprocessor for Parallel Scientific Computations," *IEEE Trans. Computers,* C-38(1):47–81, Jan. 1989.

[Hwang90] K. Hwang et al., "OMP: A RISC-based multiprocessor Using Orthogonal Access Memories and Multiple Spanning Buses," *Proc. ACM Int. Conf. Supercomputing,* pp. 7–22, Amsterdam, The Netherlands, June 1990.

[Hwang91] K. Hwang and S. Shang, "Wired-NOR Barrier Synchronization for Designing Shared-Memory Multiprocessor," *Proc. Int. Conf. Parallel Processing,* St. Charles, IL, Aug. 1991.

[Hwu91] W. M. Hwu, "Tutorial Notes on Compiler Support for Superscalar Processors," *Proc. 18th Annu. Int. Symp. Computer Arch.,* Toronto, 1991.

[Iannucci88] R. A. Iannucci, *A Dataflow/von Neumann Hybrid Architecture,* Ph.D. thesis, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1988.

[IBM90a] IBM, *RISC System/6000 Technology,* IBM Advanced Workstations Division, IBM Austin Communications Dept., Austin, TX, 1990.

[IBM90b] IBM, *System/390 Processors, System Functions,* International Business Machines, White Plains, NY, 1990.

[IEEE85] IEEE, *Standard* 754, *Order No. CN-953,* IEEE Computer Society Press, Los Alamitos, CA, 1985.

[IEEE91] IEEE, *Futurebus+: Logical Layer Specifications, 896.1-1991,* Microprocessor Standards Subcommittee,s IEEE Computer Society, 1991.

[Intel84] Intel, *Multibus II Bus Architecture Specification Handbook,* Intel Corporation, Santa Clara, CA, no. 146077-c, 1984.

[Intel90] Intel, *Supercompilers for the iPSC/860, Technical Summary,* Intel Scientific Computers, Beaverton, OR, 1990.

[Intel91] Intel, *Paragon XP/S Product Overview,* Supercomputer Systems Division, Intel Corporation, Beaverton, OR 97006, 1991.

[James90a] D. D. James, A. T. Laundrie, S. Gjessing, and G. S. Sohni, "Scalable Coherence Interface," *IEEE Computer,* 23(6):74–77, 1990.

[James90b] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi, "Distributed-Directory Scheme: Scalable Coherent Interface," *IEEE Computer,* 23(6):74–77, 1990.

[Jermoluk90] T. Jermoluk, *Multiprocessor UNIX,* Silicon Graphics Inc., Santa Clara, CA, 1990.

[Johnson91] M. Johnson, *Superscalar Microprocessor Design,* Prentice-Hall, Englewood Cliffs, NJ,1991.

[Johnsson90] L. Johnsson, "Communication in Network Architectures," in Suaya and Birtwistle (eds.), *VLSI and Parallel Computation,* Morgan Kaufmann, San Mateo, CA, 1990.

[Jones80] A. K. Jones and P. Schwarz, "Experience Using Multiprocessor Systems—A Status Report," *ACM Computing Survey*, 12(2):121–165, 1980.

[Jones86] M. B. Jones and R. F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," *Proc. OOPSLA* 1986, pp. 67–77, Portland, OR, Sept. 1986.

[Jordan83] H. F. Jordan, "Performance Measurement on HEP—A Pipelined MIMD Computer," *Proc. 10th Symp. Computer Arch.*, pp. 207–212, 1983.

[Jordan86] H. F. Jordan, "Structuring Parallel Algorithms in an MIMD, Shared Memory Environment," *Parallel Computing*, pp. 93–110, May 1986.

[Jouppi89] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proc. Third Int. Conf. Arch. Support for Prog. Lang. and OS*, pp. 272–282, ACM Press, New York, 1989.

[Kallstrom88] M. Kallstrom and S. S. Thakkar, "Programming Three Parallel Computers," *IEEE Software*, pp. 11–22, Jan. 1988.

[Kane88] G. Kane, *MIPS R2000 RISC Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Karp66] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J. Appl. Math.*, pp. 1390–1411, Nov. 1966.

[Karp88] R. M. Karp and V. Ramachandran, "A Survey of Complexity of Algorithms for Shared-Memory Machines," Technical Report 408, University of California, Berkeley, 1988.

[Katz90] R. H. Katz and J. L. Hennessy, "High-Performance Microprocessor Architectures," *Int. J. High-Speed Electronics*, 1(1), Jan. 1990.

[Kawabe87] S. Kawabe et al., "The Single Vector-Engine Supercomputer S-820," *Nikkei Electronics*, pp. 111–125, 1987.

[Kermani79] P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Communication Switching Technique," *Computer Networks*, 3(4):267–286, 1979.

[Kodama90] Y. Kodama, S. Sakai, and Y. Yamaguchi, "A Prototype of Highly Parallel Dataflow Machine EM-4 and Its Preliminary Evaluation," *Prof. Info., Japan*, 1990.

[Kogge81] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, New York, 1981.

[Kowalik85] J. S. Kowalik (ed.), *Parallel MIMD Computation: HEP Supercomputer and Applications*, MIT Press, Cambridge, MA, 1985.

[Kruatrachue88] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software*, 5(1):23–31, Jan. 1988.

[KSR91] KSR, *KSR-1 Overview*, Internal Report, Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA 02154, 1991.

[Kuck78] D. J. Kuck., *The Structure of Computers and Computations*, Wiley and Sons, New York, 1978.

[Kuck82] D. J. Kuck and R. A. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Trans. Computers*, pp. 363–376, May 1982.

[Kuck84] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Retargetable Vectorizer," in Hwang (ed.), *Supercomputers: Design and Applications*, IEEE Computer Society Press, Los Alamitos, CA, 1984.

[Kuck86] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel Supercomputing Today-The Cedar Approach," *Science*, 231(2), Feb. 1986.

[Kumar87] V. Kumar and N. Rao, "Parallel Depth-First Search, Part II: Analysis," *Int. J. Para. Programming*, 16(6):501–519, 1987.

[Kumar88] M. Kumar, "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications," *IEEE Trans. Computers*, 37(9):1088–1098, 1988.

[Kumar90] V. Kumar and V. Singh, "Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem," *Proc. Int. Conf. Parallel Processing,* pp. 136–140, 1990.

[Kumar92] V. Kumar and A. Gupta, "Analyzing Scalability of Parallel Algorithms and Architectures," Technical Report AHPCRC 92-020, Army High-Performance Computing Research Center, University of Minnesota, Minneapolis, MN, 1992.

[Kung78] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," Duff and Stewart (eds.), *Sparse Matrix Proceedings,* Knoxville, TN, 1978, SIAM, Philadelphia.

[Kung80] H. T. Kung, "The Structure of Parallel Algorithms," in Yovits (ed.), *Advances in Computers,* vol. 19, pp. 65–112, Academic Press, New York, 1980.

[Kung84] S. Y. Kung, "On Supercomputing with Systolic and Wavefront Array Processors," *Proc. IEEE,* pp. 867–884, July 1984.

[Kung88] S. Y. Kung, *VLSI Array Processors,* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Kung90] H. T. Kung, "How to Move Parallel Processing into the Mainstream," *Proc. First Workshop on Parallel Processing,* Taiwan, China, Dec. 1990.

[Lam88] M. S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implementation,* pp. 318–328, 1988.

[Lam92] M. S. Lam, "Tutorial on Compilers for Parallel Machines," Western Institute of Computer Science, Stanford University, 1992.

[Lamport78] L. Lamport, "Time, Clock, and Ordering of Events in a Distributed System," *Commun. ACM,* July 1978.

[Lamport79] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Computers,* 28(9):241–248, 1979.

[Lan90] Y. Lan, A. H. Esfahanian, and L. M. Ni, "Multicast in Hypercube Multiprocessors," *J. Para. Distri. Computing,* pp. 30–41, Jan. 1990.

[Lang82] T. Lang, M. Valero, and I. Alegre, "Bandwidth Analysis of Crossbar and Multiple-Bus Contentions for Multiprocessors," *IEEE Trans. Computers,* pp. 1227–1233, Jan. 1982.

[Larson73] A. G. Larson, "Cost-Effective Processor Design with an Application to FFT," Technical Report SU-SEL-73-037, Stanford University, Aug. 1973.

[Larson84] J. L. Larson, "Multitasking on the Cray X-MP-2 Multiprocessor," *IEEE Computer,* pp. 62–69, July 1984.

[Laudon92] J. Laudon, A. Gupta, and M. Horowitz, "Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors," Technical Report CSL-TR-92-523, Computer Systems Laboratory, Stanford University, Stanford, CA 94305-4055, 1992.

[Lawrie75] D. H. Lawrie, "Access and Alignment of Data in a Array Processor," *IEEE Trans. Computers,* Dec. 1975.

[Leasure90] B. Leasure, *PCF Fortran Extension,* Kuck & Associates, Champaign, IL 61820, 1990.

[Lee80] R. B. Lee, "Empirical Results on the Speedup, Efficiency, Redundancy, and Quality of Parallel Computations," *Proc. Int. Conf. Parallel Processing,* pp. 91–96, Aug. 1980.

[Lee84] J. K. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer,* 17(1):6–22, 1984.

[Leiserson85] C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Trans. Computers,* 34:892–901, 1985.

[Leiserson92] C. E. Leiserson et al., "The Network Architecture of the Connection Machine CM-5," *Proc. ACM Symp. Parallel Algorithms and Architecture,* San Diego, CA, 1992.

[Lenoski90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. 17th Annu. Int. Symp. Computer Arch.,* pp. 148–159, 1990.

[Lenoski92] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford Dash Multiprocessor," *IEEE Computer.* pp. 63–79, Mar. 1992.

[Lewis92] T. G. Lewis and H. El-Rewini, *Introduction to Parallel Computing,* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Li85] K. C. Li and H. Schwetman, "Vectorizing C: A Vector Processing Language," *J. Para. Distri. Computing,* 2(2):132–169, May 1985.

[Li86] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," Technical report, Yale University, 1986.

[Li88] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," *Proc. Int. Conf. Parallel Processing,* pp. 94–101, 1988.

[Li89] K. Li and P. Hudak, "Memory Coherence in Shared-Memory Systems," *ACM Trans. Computer Systems,* pp. 321–359, Nov. 1989.

[Li91] K. Li and K. Petersen, "Evaluation of Memory System Extensions," *Proc. 18th Annu. Int. Symp. Computer Arch.,* Toronto, 1991.

[Li92] K. Li, "Scalability Issues of Shared Virtual Memory for Multiprocessors," in Dubois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors,* Kluwer Academic Publishers, Boston, MA, 1992.

[Lilja88] D. J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," *IEEE Computer,* 21(7):47–55, 1988.

[Lilja92] D. J. Lilja, *Architectural Alternatives for Exploiting Parallelism,* IEEE Computer Society Press, Los Alamitos, CA, 1992.

[Lin91a] X. Lin, P. K. McKinley, and L. M. Ni, "Performance Evaluation of Multicast Wormhole Routing in 2D-Mesh Multicomputers," *Proc. Int. Conf. Parallel Processing,* vol. I, pp. 435–442, 1991.

[Lin91b] X. Lin and L. M. Ni, "Deadlock-Free Multicast Wormhole Routing in Multicomputer Networks," *Proc. 18th Annu. Int. Symp. Computer Arch.,* pp. 116–125, 1991.

[Linder91] D. H. Linder and J. C. Harden, "An Adaptive and Fault Tolerant Wormhole Routing Strategy for *k*-ary *n*-Cubes," *IEEE Trans. Computers,* 40(1):2–12, Jan. 1991.

[Margulis90] N. Margulis, *i860 Microprocessr Architecture,* Intel Osborne/McGraw-Hill, Berkeley, CA, 1990.

[Marson88] M. A. Marson, G. Balbo, and G. Conte *Performace Modules of Multiprocessor Systems,* MIT Press, Cambridge, MA, Cambridge, MA, 1988.

[MasPar91] MasPar "The MasPar Family Data-Parallel Computer", Technical summary MasPar Computer Corporation, Sunnyvale, CA, 1991.

[Mirapuri92] S. Mirapuri, M. Woodacre, and N. Vasseghi, "The MIPS R4000 Processor," *IEEE Micro,* 12(2):10–22, Apr. 1992.

[Mowry91] T. Mowry and A. Gupta, "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *J. Para. Distri. Computing* 12:87–106, June 1991.

[Muchnick88] S. S. Muchnick, "Optimizing Compilers for SPARC," *Sun Technology,* Summer:64–77, 1988.

[Mudge87] T. N. Mudge, J. P. Hayes, and D. D. Winsor, "Multiple Bus Architectures," *IEEE Computer,* 20(6):42–49, 1987.

[Nassi87] I. R. Nassi. "A Preliminary Report on the Ultramax: A Massively Parallel Shared-Memory Multiprocessor," Technical Report ETR 87-4, Encore Computer Corporation, Fort Lauderrdale, FL, 1987. Fort Lauderdale, FL, 1987.

[nCUBE90] nCUBE, *nCUBE 6400 processor Manual, n*CUBE Company, Beavrton, OR 97006, 1990.

[NEC90] NEC, "SX-X Series", HNSX," Technical report, Nippon Electric Company, Japan, 1990.

[NeXT90] NeXT Computer, Inc., Redwood City, CA, *The Mach Operating System, Chapter 1,* 1990.

[Ni85a] L. M. Ni and K. Hwang, "Optimal Load Balancing in a Multiple Processor System with Many Job Classes," *IEEE Trans. Software Engineering,* pp. 491–496, May 1985.

[Ni85b] L. M. Ni and K. Hwang, "Vector Reduction Techniques for, Arithmetic Pipeline," *IEEE Trans. Computers*, pp. 404–411, May 1985.

[Ni91] L. M. Ni, "A Layered Classification of Parallel Computers," *Proc. 1991 Int. Conf. for Young Computer Scientists*, pp. 28–33, Beijing, China, May 1991.

[Nickolls90] J. R. Nickolls, "The Design of the MasPar MP-1: A Cost-Effective Massively Parallel Computer," in *IEEE Digest of Papers-Comcom*, pp. 25–28, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Nicol88] D. M. Nicol and F. H. Willard, "Problem Sise, Parallel Architecture, and Optimal Speedup," *J. Para. Distri. Computing*, 5:404–420, 1988.

[Nicolau84] A Nicolau and J. A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Trans. Computers*, 33(11):968–976, 1984.

[Nikhil89] R. S. Nikhil and Arvind, "Can Dataflow Subsume von Neumann Computing?," *Proc. 16th Annu. Int. Symp. Computer Arch.*, pp· 262–272, 1989.

[Nikhil92a] R. S. Nikhil, "'Tutorial Notes on Multithreaded Architectures," *Proc. 19th Annu. Inl. Symp. Computer Arch.*, 1992, Contact DEC Cambridge Res,. Lab., 1 Kendall Square, Bldg. 700, Cambridge, MA 02139.

[Nikhil92b] R. S. Nikhil and G. M. Papadopoulos, "*T: A Multithreaded Massively Parallel Architecture," *Proc. 19th Annu. Int. Symp. Computer Arch.*, May 1992.

[Nitzberg91] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, 24(8):52–60, 1991.

[Noakes90] M. Noakes and W. J. Dally, "System Design of the J-Machine," in Dally (ed.), *Proc. Sixth MIT Conf. Advanced Research in VLSI*, pp. 179–194, MIT Press, Cambridge, MA,1990.

[NS88] NS, *NS32532 Performance Analysis: A Benchmark Study*, National Semiconductor, 1988.

[NSF92] NSF, "Grand Challenge: High-Performance Computing and Communications," Report, Committee on Physical, Mathematical, and Engineering Sciences, U.S. Office of Science and Technology Policy, National Science Foundation, Washington, DC, 1992.

[Nussbaum91] D. Nussbaum and A. Agarwal, "Scalability of Parallel Machines," *Commun. ACM*, 34(3):57–61, 1991.

[OSF90] OSF, *OSF/1 Technical Seminar*, Open Software Foundation, Inc., Cambridge, MA, 1990.

[Ousterhout88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *IEEE Computer*, 21(2):23–36, 1988.

[Padua80] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. Computers*, pp. 763–776, Sept. 1980.

[Padua86] D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Commun. ACM*, pp. 1184–1201, Dec. 1986.

[Panda91] D. K. Panda and K. Hwang, "Fast Data Manipulation in Multiprocessors Using Parallel Pipelined Memories," *J. Para. Dislri. Computing*, 12:130–145, June 1991.

[Parasoft90] Parasoft , *Express User's Guide Version 3.0*, Parasoft Corporation, Pasadena, CA 90025, 1990.

[Parker91] K. Parker, "The Next Generation Furturebus+," *Futurebus+ Design*, (1):12–28, Jan. 1991.

[Patel78] J. H. Patel, "Pipelines with Internal Buffers," *Proc. 5th Symp. Computer Arch.*, pp. 249–254, 1978.

[Patel81] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Trans. Computers*, pp. 771–780, Oct. 1981.

[Patel82] J. H. Patel, "Analysis of Multiprocessors with Private Caches," *IEEE Trans. Computers*, C-31(4):296–304, Apr. 1982.

[Patterson82] D. Patterson and C. Sequin, "A VLSI RISC," *IEEE Computer*, 15(9), 1982.

[Perrott79] R. H. Perrott, "A Language for Array and Vector Processors," *ACM Trans. Prog. Lang. and Systems,* 1(2):177–195, Oct. 1979.

[Perrott87] R. H. Perrott, *Parallel Programming,* Addison-Wesley, Reading, MA, 1987.

[Pfister85a] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norlton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. Int. Conf. Parallel Processing,* pp. 764–771, Aug. 1985.

[Pfister85b] G. F. Pfister and V. A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," *Proc. Int. Conf. Parallel Processing,* pp. 790–797, Aug. 1985.

[Polychronopoulos89] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase 2: An Environment for Parallelizing, Partitioning, Synchronizing Programs on Multiprocessors," *Proc. Int. Conf. Parallel Processing,* pp. 765–777, 1989.

[Pountain87] D. Pountain and D. May, *A Tutorial Introduction to Occam Programming,* McGraw-Hill, New York, 1987.

[Prasanna Kumar87] V. K. Prasanna Kumar and C. S. Raghavendra, "Array Processor with Multiple Broadcasting," *J. Para. Distri. Computing,* pp. 1202–1206, Apr. 1987.

[Preparata79] F. P. Preparata and J. E. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Proc. 20th Symp. Foundations Computer Sci.,* pp, 140–147, 1979.

[Przybylski90] S. Przybylski, *Cache and Memory Hierarchy Design,* Morgan Kaufmann, San Mateo, CA, 1990.

[PSR90] PSR, *MIMDizer User's Guide Version 7.01,* Pacific Science Research, Placerville, CA 90025, 1990.

[Quinn87] M. J. Quinn, *Designing Efficient Algorithms for Parallel Commuters,* McGraw-Hill, New York, 1987.

[Quinn90] M. J. Quinn and P. J. Hatcher, "Data-Parallel Programming on Multicomputers," *IEEE Software,* 7(5):69–76, Sept. 1990.

[Ragsdale90] S. Ragsdale (ed.), *Parallel Programming Primer,* Intel Scientific Computers, Beaverton, OR, 1990.

[Ramamoorthy77] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," *ACM Computing Survey,* pp. 61–102, Mar. 1977.

[Rashid81] R. F. Rashid and G. G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel," *Proc. 8th ACM Symp. Operating System. Principles,* pp. 64–75, Dec. 1981.

[Rashid86] R. F. Rashid, "From RIG to Accent to Mach: The Evolution of a Network Operating System," *Proc. Fall Joint Computer Conf.,* pp. 1128–1137, Dallas, TX, Nov. 1986.

[Rice85] J. R. Rice, "Problems to Test Parallel and Vector Languages," Technical Report CSD-TR 516, Purdue University, May 1985.

[Ritchie74] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Commun. ACM,* 17(7):365–375, July 1974.

[Rothnie91] J. Rothnie, "KSR-1 Memory System," Technical report, Kendall Square Research, Cambridge, MA, 1991.

[Russell87] C. H. Russell and P. J. Waterman, "Variations on UNIX for Parallel Processing Computers," *Commun. ACM,* 30(12):1048–1055, Dec. 1987.

[Saavedra90] R. H. Saavedra, D. E. Culler, and T. von Eicken, "Analysis of Multithreaded Architectures for Parallel Computing," *Proc. ACM Symp. Parallel Algorithms and Architecture,* Greece, July 1990.

[Sakai91] S. Sakai, Y. Kodama, and Y. Yamaguchi, "Prototype Implementation of a Highly Parallel Dataflow Machine EM-4," *Proc. Int. Parallel Processing Symposium,* 1991

[Sayanarayanan80] M. Sayanarayanan, "Commercial Multiprocessing Systems," *IEEE Computer,* 13(5):75–96, 1980.

[Scheurich89] C. Scheurich, *Access Ordering and Coherence in Shared-Memory Multiprocessors,* Ph.D. thesis, University of Southern California, 1989.

[Schimmel90] C. Schimmel, "UNIX on Modern Architectures," *Proc. Summer 1990 USENIX Conf.*, Anaheim, CA, June 1990.

[Schwartz80] J. T. Schwartz, "Ultra-Computers," *A CM Trans. Prog. Lang. and Systems*, 2(4):484–521,1980.

[SCSI84] Computer Business Equipment Manufacturers, Wash. DC, *SCSI Small Computer System Interface, ANSC X3*, 1984.

[Seitz85] C. L. Seitz, "The Cosmic Cube," *Commun. ACM*, 28(1), 1985.

[Seitz88] C. L. Seitz, W. C. Athas, C. M. Flaig, A. J. Martin, J. Seizovic, C. S. Steele, and W. K. Su, "The Architecture and Programming of the Ametek Series 2010 Multicomputer," *Proc. Conf. Hypercube Computers and Concurrent Applications*, pp. 33–36, pasadena, CA, Jan. 1988.

[Seitz89] C. L. Seitz, J. Seizovic, and W. K. Su, "The C-Programmer's Guide to Multicomputer Programming," Technical Report CS-TR-88-1, California Institute of Technology, Pasadena, CA, 1989.

[Seitz90] C. L. Seitz, "Concurrent Architectures," in Suaya and Birtwistle (eds.), *VLSI and Parallel Computation*, Chapter 3, Morgan Kaufmann, San Mateo, CA, 1990.

[Seitz92] C. L. Seitz, "Mosaic C: An Experimental Fine-Grain Multicomputer," Technical report, California Institute of Technology, Pasadena, CA 91125, 1992.

[Sevcik89] K. Sevcik, "Characterization of Parallelism in Applications and Their Use in Scheduling," *Proc. ACM SIGMETRICS and Performance*, May 1989.

[Shapiro86] E. Shapiro, "Concurrent Prolog," *IEEE Computer*, 19(1):44–58, 1986.

[Shar72] L. E. Shar, "Design and Scheduling of Statistically Configured Pipelines," Lab Report SU-SEL-72-042, Stanford University, 1972.

[Sheperdson63] J. C. Sheperdson and H. E. Sturgis, "Computability of Recursive Functions," *J. ACM.* 10:217–255, 1963.

[Shih89] Y. Shih and J. Fier, "Hypercube Systems and Key Applications," in Hwang and DeGroot (eds.), *Parallel Processing for Supercomputing and Artificial Intelligence*. pp. 203–244, McGraw-Hill, New York, 1989.

[Siegel79] H. J. Siegel, "A Model of SIMD Machines and a Comparison of Various Interconnection Networks," *IEEE Trans. Computers*, 28(12):907–917,1979.

[Siegel89] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, 2nd ed., McGraw-Hill, New York, 1989.

[Siewiorek91] D. P. Siewiorek and P. J. Koopman, *The Architecture of Supercomputers, TITAN: A Case Study*, Academic Press, New York, 1991.

[Simmons92] M. L. Simmons, H. J. Wasserman, O. M. Lubeck, C. Eoyang, R. Mendez, H. Harada, and M. Ishiguro, "A Performance Comparison of Four Supercomputers," *Commun. ACM*, 35(8):116–124, 1992.

[Sindhu92] P. S. Sindhu, J. M. Frailong, and M. Cekleov, "Formal Specification of Memory Modules," in Dubois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors*, Kluwer Academic publishers, Boston, MA, 1992.

[Smith82] A. J. Smith, "Cache Memories," *ACM Computing Survey*, pp. 473–530, Sept. 1982.

[Smith85] B. Smith, "The Architecture of the HEP," in Kowalik (ed.), *Parallel MIMD Computation: HEP Supercomputer and Applications*, MIT Press, Cambridge, MA, 1985.

[Smith88] J. E. Smith, "Characterizing Computer Performance with a Single Number," *Commun. ACM*, 31(10): 1202–1206, 1988.

[Smith89] J. E. Smith, "Dynamic Instruction Scheduling and The Astronautics ZS-I," *IEEE Computer*, 22(7):21–35, 1989.

[Smith90] J. E. Smith, W. C. Hsu, and C. Hsiung, "Future General-Purpose Supercomputer Architecture," *Proc. ACM Supercomputing Conf. 1990*, New York, Nov. 1990.

[Snir82] M. Snir, "On Parallel Search," *Proc. Principles of Distributed Computing*, pp. 242–253,1982.

[Sohi90] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. Computers*, 39(3):349–359, March 1990.

[SPARC90] Sun Microsystems, *SPARC Architecture Reference Manual VB*, Dec. 1990.

[Stallings90] W. Stallings, *Reduced Instruction Set Computers*. 2nd ed., IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Stenström90] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, 23(6): 12–25, 1990.

[Stenström92] P. Stenström, T. Joe, and A. Gupta, "Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures," *Proc. 19th Annu. Int. Symp. Computer Arch., 1992.*

[Stone71] H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Computers*, C20:153–161, 1971.

[Stone90] H. S. Stone, *High-Performance Computer Architecture*, Addison-Wesley. Reading, MA,1990.

[Sullivan77] H. Sullivan and T. R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine," *Proc. 4th Symp. Computer Arch.*, vol. 5, pp. 105–124, Mar. 1977.

[Sun91] X. H. Sun and D. T. Rover, "Scalability of Parallel Algorithm-Machine Combinations," Technical Report IS-5057, UC-32, Ames Laboratory, Iowa State University, Ames, Iowa, 1991.

[Sun93] X. H. Sun and L. M. Ni, "Scalable Problems and Memory-Bound Speedup," *J. Para. Distri. Computing*, 1993, also appeared in *Proc. ACM Supercomputing, 1990.*

[Sweazey89] P. Sweazey, "Cache Coherence on SCI," *IEEE Computer Architecture Workshop*, Elilat, Israel, May 1989.

[Tabak90] D. Tabak, *Multiprocessors*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Tabak91] D. Tabak, *Advanced Microprocessors*, McGraw-Hill, New York, 1991.

[Tanenbaum92] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal, "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, 25(8):10–20, 1992.

[Tevanian87] A. Tevanian, R. F. Rashid, M. W. Young, D. B.' Golub, D. L. Black, and E. Cooper, "Mach Threads and the UNIX Kernel: The Battle for Control," *Proc. Summer 19B7 USENIX Con/.*, pp. 185–197, Phoenix, AZ, June 1987.

[Tevanian89] A. Tevanian and B. Smith, "Mach' The Model for Future UNIX," *Byte*, 14(12):411–416, Nov. 1989.

[Thakkar90] S. S. Thakkar, M. Dubois, A. T. Laundrie, G. S. Sohi, D. V. James, S. Gjessing, M. Thapar, B. Delagi, M. Carlton, and A. Despain, "New Directions in Scalable Shared-Memory Multiprocessor,. Architectures", *IEEE Computer*, 23(6):71–83, 1990.

[Thompson80] S. D. Thompson *A Complexity Theory for VLSI*, Ph.D. thesis, Carnegie-Mellon University, 1980.

[Thornton70] J. E. Thornton, *Design of a Computer: The CDC 6600*, Soott and Foresman, Glenview, IL, 1970.

[TI83] Texas Instruments Inc., Dallas, TX, *NuBus Specification*, 1983.

[TMC90] TMC, *The CM-2 Technical Summary*, Thinking Machines Corporation, Cambridge, MA,1990.

[TMC91] TMC, *The CM-5 Technical Summary*, Thinking Machines Corporation, Cambridge, MA,1991.

[Tomasulo67] R. M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. and Develop.*, 11(1):25–33, 1967.

[Treleaven85] P. C. Treleaven, "Control-Driven, Data-Driven, and Demand-Driven Computer Architecture," *Parallel Computing*, 2, 1985.

[Trew91] A. Trew and G. Wilson (eds.), *Past, Present, Parallel: A Survey of Available Parallel Computer Systems*, Springer-Verlag, London, 1991.

[Tucker88] L. W. Tucker and G. G. Robertson, "Association and Applications of the Connection Machine", *IEEE Computer*, 21(8):26–38, 1988.

[Ullman84] J. D. Ullman, *Computational Aspects of VLSI,* Computer, Science Press, Rockville, MD,1984.

[VITA90] VME Bus International Trade Association and IEEE P1014 Working Group, *64-Bit VMEbus Specification, Edition* D, Jan. 1990.

[Wah90] B. W. Wah and C. V. Ramamoorthy (eds.) , *Computers for Artificial Intelligence Processing,* Wiley, New York, 1990.

[Wall91] D. W. Wall, "Limits of Instruction-Level Parallelism," *Proc. Fourth Int. Conf. Arch. Support for Prog. Lang. and OS,* pp. 176–188, 1991.

[Wallace64] C. E. Wallace "A Suggestion for Fast Multiplier ," *IEEE Trans. Computers* pp. 14–17, Feb. 1964.

[Waltz87] D. L. Waltz, "Applications of the Connection Machine. (New Computer Architecture form Thinking Machines Corporation, *IEEE Computer,* 20(1):85–97,1987.

[Wang92] H. C. Wang, *Parallelization of Iteraive PDE Solvers on Shared-Memory Multiprocessors,* Ph.D. thesis, University of Southern California, 1992.

[Wang93] H. C. Wang and K. Hwang, "Multicoloring Parallelization of Grid-Structured PDE Solvers", Technical report, University of Southern California, Los Angels, 1993.

[Weicker84] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark", *Commun. ACM,* 27(10): 1013–1030, 1984.

[Weiser85] W. Weiser et al., "Status and Performance of the Zmob Parallel Processing Systems," *Proc. COMPCON,* pp. 71–73, 1985.

[Weiss84] S. Weiss and J. E. Smith, "Instruction Issue Logic in Pipelined Supercomputers," *IEEE Trans. Computers,* pp. 1013–1022, 1984.

[Wilson87] A. W. Wilson, "Hierarchical Cache/Bus Architecture for Shared-Memory Multiprocessors," *Proc. 14th Annu. Int. Symp. Computer Arch.,* pp. 244–252, 1987.

[Wirth77] N. Wirth, "Modula: A Language for Modular Multiprogramming," *Software Practice and Experience,* 7:3–35, Jan. 1977.

[Wolf91a] M. E. Wolf and M. S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. Parallel Distri. Systems,* 3(10):452–471, 1991.

[Wolf91b] M. E. Wolf and M. S. Lam, "A Data Locality Optimization Algorithm," *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implementation,* pp. 30–44, 1991.

[Wolfe82] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers,* Ph.D. thesis, University of Illinois, 1982.

[Wolfe89] M. J. Wolfe, "Automatic Vectorization, Data Dependence, and Optimizations for Parallel Computers," in Hwang and DeGroot (eds.), *Parallel Processing for Supercomputing and Artificial Intelligence,* Chapter 11, McGraw-Hill, New York, 1989.

[Worlton84] J. Worlton, "Understanding Supercomputer Benchmarks," *Automation,* pp. 121–130, Sept. 1984.

[Wu80] C. L. Wu and T. Y. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Trans. Computers,* pp. 696–702, Aug. 1980.

[Wulf72] W. A. Wulf and C. G. Bell, "C.mmp—A Multi-Miniprocessor," *Proc. Fall Joint Computer Conf.,* pp. 765–777, 1972.

[Xu89] Z. Xu and K. Hwang, "Molecule: A Language Construct for Layered Development of Parallel Programs," *IEEE Trans. Computers,* 38(5):587–599, 1989.

[Xu91] J. Xu and K. Hwang, "Mapping Rule-Based Systems onto Multicomputers using Simulated Annealing," *J. Para. Distri. Computing,* pp. 442–455, Dec. 1991.

[Yamaguchi91] Y. Yamaguchi, S. Sakai, and Y. Kodama, "Synchronization Mechanisms of a Highly Parallel Dataflow Machine EM-4," *IEICE Trans.,* 74(1):204–213, 1991.

[Yew87] P. C. Yew, N. F. Tseng, and D. H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Trans. Computers,* pp. 388–395, Apr. 1987.

[Yew91] P. C. Yew and B. W. Wah (eds.), Special Issue on Shared-Memory Multiprocessors, *J. Para. Distri. Computing,* June 1991.

[Young87] M. W. Young, A. Tevanian, R. F. Rashid, D. B. Golub, J. Eppinger, J. Chew, W. Bolosky, D. L. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc. 11th ACM Symp. Operating System Principles,* pp. 63–76, 1987.

[Zima90] H. Zima and B. Chapman, *Supercompiler for Parallel and Vector Computers,* Addison-Wesley, Reading, MA, 1990.

[Zorpetta92] G. Zorpetta, "The Power of Parallelism," *IEEE Spectrum,* 29(9):28–33, 1992.

Index

N-18911/3 Re-6