Computer Architecture

A Quantitative Approach

THIRD EDITION

John L. Hennessy

David A. Patterson

Computer Architecture
A Quantitative Approach

Third Edition

John L. Hennessy
Stanford University

David A. Patterson
University of California at Berkeley

With Contributions by

David Goldberg
Xerox Palo Alto Research Center

Krste Asanovic
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER SCIENCE

AMSTERDAM BOSTON LONDON NEW YORK
OXFORD PARIS SAN DIEGO SAN FRANCISCO
SINGAPORE SYDNEY TOKYO

Senior Editor Denise E. M. Penrose
Assistant Publishing Services Manager Edward Wade
Senior Production Editor Cheri Palmer
Editorial Coordinator Alyson Day
Cover Design Ross Carron Design
Cover Image Greg Pease / gettyimages
Text Design Rebecca Evans & Associates
Technical Illustration Lineworks, Inc.
Composition Nancy Logan
Copyeditor Ken DellaPenta
Proofreader Jennifer McClain
Indexer Ty Koontz
Printer Courier Corporation

Morgan Kaufmann Publishers
An Imprint of Elsevier Science
340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA
www.mkp.com

Published 1990. Third edition 2003
Printed in the United States of America
07 06 05 04 03 10 9 8 7 6 5 4 3 2 1

ADVICE, PRAISE, & ERRORS: Any correspondence related to this publication or intended for the authors should be addressed to *ca3comments@mkp.com*. Information regarding error sightings is also encouraged. Any error sightings that are accepted for correction in subsequent printings will be rewarded by the authors with a payment of $1.00 (U.S.) per correction upon availability of the new printing. Bugs can be sent to *cabugs@mkp.com*. (Please include your full name and permanent mailing address.)

Library of Congress Control Number: 2001099789

ISBN 1-55860-596-7 (cloth)
ISBN 1-55860-724-2 (paper)

This book is printed on acid-free paper.

6.1 Introduction

As the quotations that open this chapter show, the view that advances in uniprocessor architecture were nearing an end has been widely held at varying times. To counter this view, we observe that during the period 1985–2000, uniprocessor performance growth, driven by the microprocessor, was at its highest rate since the first transistorized computers in the late 1950s and early 1960s.

On balance, though, we believe that parallel processors will definitely have a bigger role in the future. This view is driven by three observations. First, since microprocessors are likely to remain the dominant uniprocessor technology, the logical way to improve performance beyond a single processor is by connecting multiple microprocessors together. This combination is likely to be more cost-effective than designing a custom processor. Second, it is unclear whether the pace of architectural innovation that has been based for more than 15 years on increased exploitation of instruction-level parallelism can be sustained indefinitely. As we saw in Chapters 3 and 4, modern multiple-issue processors have become incredibly complex, and the performance increases achieved through increasing complexity, increasing silicon, and increasing power seem to be diminishing. Third, there appears to be slow but steady progress on the major obstacle to widespread use of parallel processors, namely, software. This progress is probably faster in the server and embedded markets, as we discussed in Chapters 3 and 4. Server and embedded applications exhibit natural parallelism that can be exploited without some of the burdens of rewriting a gigantic software base. This is more of a challenge in the desktop space.

We, however, are extremely reluctant to predict the death of advances in uniprocessor architecture. Indeed, we believe that the rapid rate of performance growth will continue at least for the next five years. Whether this pace of innovation can be sustained longer is difficult to predict but hard to bet against. Nonetheless, if the pace of progress in uniprocessors does slow down, multiprocessor architectures will become increasingly attractive.

That said, we are left with two problems. First, multiprocessor architecture is a large and diverse field, and much of the field is in its youth, with ideas coming and going and, until very recently, more architectures failing than succeeding. Given that we are already on page 528, full coverage of the multiprocessor design space and its trade-offs would require another volume. (Indeed, Culler, Singh, and Gupta [1999] cover *only* multiprocessors in their 1000-page book!) Second, such coverage would necessarily entail discussing approaches that may not stand the test of time, something we have largely avoided to this point. For these reasons, we have chosen to focus on the mainstream of multiprocessor design: multiprocessors with small to medium numbers of processors (≤ 128). Such designs vastly dominate in terms of both units and dollars. We will pay only slight attention to the larger-scale multiprocessor design space (≥ 128 processors). At the present, the future architecture of such multiprocessors is unsettled, and even the viability of that marketplace is in doubt. We will return to this topic briefly at the end of the chapter, in Section 6.15.

A Taxonomy of Parallel Architectures

We begin this chapter with a taxonomy so that you can appreciate both the breadth of design alternatives for multiprocessors and the context that has led to the development of the dominant form of multiprocessors. We briefly describe the alternatives and the rationale behind them; a longer description of how these different models were born (and often died) can be found in the historical perspective at the end of the chapter.

The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. About 30 years ago, Flynn [1966] proposed a simple model of categorizing all computers that is still useful today. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor, and placed all computers into one of four categories:

1. *Single instruction stream, single data stream* (SISD)—This category is the uniprocessor.
2. *Single instruction stream, multiple data streams* (SIMD)—The same instruction is executed by multiple processors using different data streams. Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions. The multimedia extensions we considered in Chapter 2 are a limited form of SIMD parallelism. Vector architectures are the largest class of processors of this type.
3. *Multiple instruction streams, single data stream* (MISD)—No commercial multiprocessor of this type has been built to date, but may be in the future. Some special-purpose stream processors approximate a limited form of this (there is only a single data stream that is operated on by successive functional units).
4. *Multiple instruction streams, multiple data streams* (MIMD)—Each processor fetches its own instructions and operates on its own data. The processors are often off-the-shelf microprocessors.

This is a coarse model, as some multiprocessors are hybrids of these categories. Nonetheless, it is useful to put a framework on the design space.

As discussed in the historical perspectives, many of the early multiprocessors were SIMD, and the SIMD model received renewed attention in the 1980s, and except for vector processors, was gone by the mid-1990s. MIMD has clearly emerged as the architecture of choice for general-purpose multiprocessors. Two factors are primarily responsible for the rise of the MIMD multiprocessors:

1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user multiprocessors focusing on high performance for one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of these functions.

2. MIMDs can build on the cost-performance advantages of off-the-shelf microprocessors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers.

With an MIMD, each processor is executing its own instruction stream. In many cases, each processor executes a different process. Recall from the last chapter that a process is a segment of code that may be run independently, and that the state of the process contains all the information necessary to execute that program on a processor. In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of the processes on other processors.

It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space. When multiple processes share code and data in this way, they are often called *threads*. Today, the term *thread* is often used in a casual way to refer to multiple loci of execution that may run on different processors, even when they do not share an address space.

To take advantage of an MIMD multiprocessor with *n* processors, we must usually have at least *n* threads or processes to execute. The independent threads are typically identified by the programmer or created by the compiler. Since the parallelism in this situation is contained in the threads, it is called *thread-level parallelism*.

Threads may vary from large-scale, independent processes—for example, independent programs running in a multiprogrammed fashion on different processors—to parallel iterations of a loop, automatically generated by a compiler and each executing for perhaps less than a thousand instructions. Although the size of a thread is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction is that such parallelism is identified at a high level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel. In contrast, instruction-level parallelism is identified primarily by the hardware, although with software help in some cases, and is found and exploited one instruction at a time.

Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictate a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization because what constitutes a small or large number of processors is likely to change over time.

The first group, which we call *centralized shared-memory architectures,* has at most a few dozen processors in 2000. For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory and to interconnect the processors and memory by a bus. With large caches, the bus and the single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors. By replacing a single bus with multiple buses, or even a switch, a centralized shared-memory design can be scaled to a few dozen processors. Although scaling beyond that is technically conceivable, sharing a centralized memory, even organized as multiple banks, becomes less attractive as the number of processors sharing it increases.

Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are often called *symmetric (shared-memory) multiprocessors* (SMPs), and this style of architecture is sometimes called *uniform memory access* (UMA). This type of centralized shared-memory architecture is currently by far the most popular organization. Figure 6.1 shows what these multiprocessors look like. The architecture of such multiprocessors is the topic of Section 6.3.

The second group consists of multiprocessors with physically distributed memory. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the scale of multiprocessor for which distributed memory is preferred over a single, centralized memory continues to decrease in number (which is another reason not to use small and large scale). Of course, the larger number of processors raises the need for a high bandwidth interconnect, of which we will see examples in Chapter 8. Both direct interconnection networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used. Figure 6.2 shows what these multiprocessors look like.

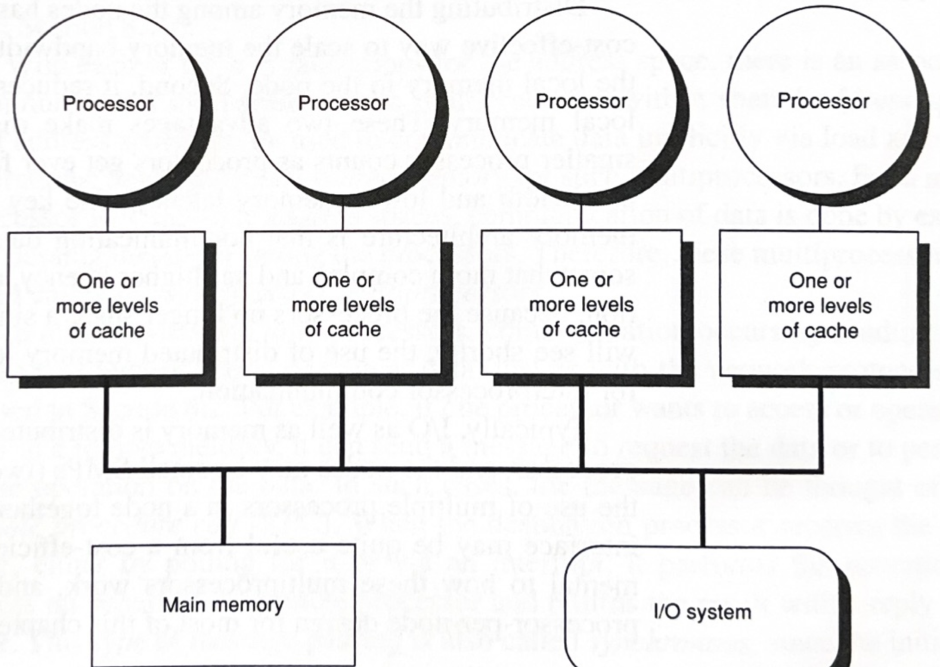
Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

Figure 6.1 Basic structure of a centralized shared-memory multiprocessor. Multiple processor-cache subsystems share the same physical memory, typically connected by a bus. In larger designs, multiple buses, or even a switch may be used, but the key architectural property—uniform access time to all memory from all processors—remains.

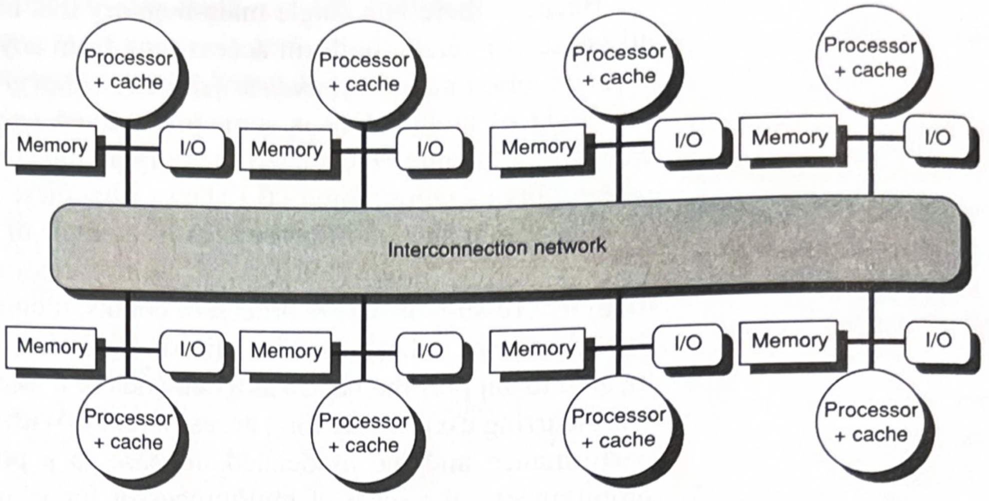

Figure 6.2 The basic architecture of a distributed-memory multiprocessor consists of individual nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes. Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network.

Distributing the memory among the nodes has two major benefits. First, it is a cost-effective way to scale the memory bandwidth if most of the accesses are to the local memory in the node. Second, it reduces the latency for accesses to the local memory. These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory bandwidth and lower memory latency. The key disadvantage for a distributed-memory architecture is that communicating data between processors becomes somewhat more complex and has higher latency, at least when there is no contention, because the processors no longer share a single, centralized memory. As we will see shortly, the use of distributed memory leads to two different paradigms for interprocessor communication.

Typically, I/O as well as memory is distributed among the nodes of the multiprocessor, and the nodes may be small SMPs (two to eight processors). Although the use of multiple processors in a node together with a memory and a network interface may be quite useful from a cost-efficiency viewpoint, it is not fundamental to how these multiprocessors work, and so we will focus on the one-processor-per-node design for most of this chapter.

Models for Communication and Memory Architecture

As discussed earlier, any large-scale multiprocessor must use multiple memories that are physically distributed with the processors. There are two alternative architectural approaches that differ in the method used for communicating data among processors.

In the first method, communication occurs through a shared address space. That is, the physically separate memories can be addressed as one logically shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. These multiprocessors are called *distributed shared-memory* (DSM) architectures. The term *shared memory* refers to the fact that the *address space* is shared; that is, the same physical address on two processors refers to the same location in memory. Shared memory does *not* mean that there is a single, centralized memory. In contrast to the symmetric shared-memory multiprocessors, also known as UMAs (uniform memory access), the DSM multiprocessors are also called NUMAs (nonuniform memory access), since the access time depends on the location of a data word in memory.

Alternatively, the address space can consist of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor. In such multiprocessors, the same physical address on two different processors refers to two different locations in two different memories. Each processor-memory module is essentially a separate computer; therefore, these parallel processors have been called *multicomputers*. A multicomputer can even consist of completely separate computers connected on a local area network, which today are popularly called *clusters*. For applications that require little or no communication and can make use of separate memories, such clusters of processors, whether using a standardized or customized interconnect, can form a very cost-effective approach (see Section 8.10).

With each of these organizations for the address space, there is an associated communication mechanism. For a multiprocessor with a shared address space, that address space can be used to communicate data implicitly via load and store operations; hence the name *shared memory* for such multiprocessors. For a multiprocessor with multiple address spaces, communication of data is done by explicitly passing messages among the processors. Therefore, these multiprocessors are often called *message-passing multiprocessors*.

In message-passing multiprocessors, communication occurs by sending messages that request action or deliver data, just as with the network protocols discussed in Section 8.2. For example, if one processor wants to access or operate on data in a remote memory, it can send a message to request the data or to perform some operation on the data. In such cases, the message can be thought of as a *remote procedure call* (RPC). When the destination processor receives the message, either by polling for it or via an interrupt, it performs the operation or access on behalf of the remote processor and returns the result with a reply message. This type of message passing is also called *synchronous,* since the initiating processor sends a request and waits until the reply is returned before continuing. Software systems have been constructed to encapsulate the details of sending and receiving messages, including passing complex arguments or return values, presenting a clean RPC facility to the programmer.

Communication can also occur from the viewpoint of the writer of data rather than the reader, and this can be more efficient when the processor producing data knows which other processors will need the data. In such cases, the data can be sent directly to the consumer process without having to be requested first. It is often possible to perform such message sends asynchronously, allowing the sender process to continue immediately. Often the receiver will want to block if it tries to receive the message before it has arrived; in other cases, the reader may check whether a message is pending before actually trying to perform a blocking receive. Also the sender must be prepared to block if the receiver has not yet consumed an earlier message and no buffer space is available. The message-passing facilities offered in different multiprocessors are fairly diverse. To ease program portability, standard message-passing libraries (for example, message-passing interface, or MPI) have been proposed. Such libraries sacrifice some performance to achieve a common interface.

Performance Metrics for Communication Mechanisms

Three performance metrics are critical in any communication mechanism:

1. *Communication bandwidth*—Ideally the communication bandwidth is limited by processor, memory, and interconnection bandwidths, rather than by some aspect of the communication mechanism. The bisection bandwidth (see Section 8.5) is determined by the interconnection network. The bandwidth in or out of a single node, which is often as important as bisection bandwidth, is affected both by the architecture within the node and by the communication mechanism. How does the communication mechanism affect the communication bandwidth of a node? When communication occurs, resources within the nodes involved in the communication are tied up or occupied, preventing other outgoing or incoming communication. When this occupancy is incurred for each word of a message, it sets an absolute limit on the communication bandwidth. This limit is often lower than what the network or memory system can provide. Occupancy may also have a component that is incurred for each communication event, such as an incoming or outgoing request. In the latter case, the occupancy limits the communication rate, and the impact of the occupancy on overall communication bandwidth depends on the size of the messages.

2. *Communication latency*—Ideally the latency is as low as possible. As we will see in Chapter 8,

$$\begin{aligned}\text{Communication latency} &= \text{Sender overhead} + \text{Time of flight}\\ &\quad + \text{Transmission time} + \text{Receiver overhead}\end{aligned}$$

   Time of flight is fixed and transmission time is determined by the interconnection network. The software and hardware overheads in sending and receiving messages are largely determined by the communication mechanism and its implementation. Why is latency crucial? Latency affects both performance and how easy it is to program a multiprocessor. Unless latency is hidden, it directly affects performance either by tying up processor resources or by causing the processor to wait. Overhead and occupancy are closely related, since many forms of overhead also tie up some part of the node, incurring an occupancy cost, which in turn limits bandwidth. Key features of a communication mechanism may directly affect overhead and occupancy. For example, how is the destination address for a remote communication named, and how is protection implemented? When naming and protection mechanisms are provided by the processor, as in a shared address space, the additional overhead is small. Alternatively, if these mechanisms must be provided by the operating system for each communication, this increases the overhead and occupancy costs of communication, which in turn reduce bandwidth and increase latency.

3. *Communication latency hiding*—How well can the communication mechanism hide latency by overlapping communication with computation or with other communication? Although measuring this is not as simple as measuring the first two metrics, it is an important characteristic that can be quantified by measuring the running time on multiprocessors with the same communication latency but different support for latency hiding. We will see examples of latency-hiding techniques for shared memory in Sections 6.8 and 6.10. Although hiding latency is certainly a good idea, it poses an additional burden on the software system and ultimately on the programmer. Furthermore, the amount of latency that can be hidden is application dependent. Thus, it is usually best to reduce latency wherever possible.

Each of these performance measures is affected by the characteristics of the communications needed in the application. The size of the data items being communicated is the most obvious, since it affects both latency and bandwidth in a direct way, as well as affecting the efficacy of different latency-hiding approaches. Similarly, the regularity in the communication patterns affects the cost of naming and protection, and hence the communication overhead. In general, mechanisms that perform well with smaller as well as larger data communication requests, and irregular as well as regular communication patterns, are more flexible and efficient for a wider class of applications. Of course, in considering any communication mechanism, designers must consider cost as well as performance.

Advantages of Different Communication Mechanisms

Each of these two primary communication mechanisms has its advantages. For shared-memory communication, advantages include

- Compatibility with the well-understood mechanisms in use in centralized multiprocessors, which all use shared-memory communication. The OpenMP consortium (see *www.openmp.org* for description) has proposed a standardized programming interface for shared-memory multiprocessors.
- Ease of programming when the communication patterns among processors are complex or vary dynamically during execution. Similar advantages simplify compiler design.

- The ability to develop applications using the familiar shared-memory model, focusing attention only on those accesses that are performance critical.
- Lower overhead for communication and better use of bandwidth when communicating small items. This arises from the implicit nature of communication and the use of memory mapping to implement protection in hardware, rather than through the I/O system.
- The ability to use hardware-controlled caching to reduce the frequency of remote communication by supporting automatic caching of all data, both shared and private. As we will see, caching reduces both latency and contention for accessing shared data. This advantage also comes with a disadvantage, which we mention below.

The major advantages for message-passing communication include the following:

- The hardware can be simpler, especially by comparison with a scalable shared-memory implementation that supports coherent caching of remote data.
- Communication is explicit, which means it is simpler to understand; in shared-memory models, it can be difficult to know when communication is occurring and when it is not, as well as how costly the communication is.
- Explicit communication focuses programmer attention on this costly aspect of parallel computation, sometimes leading to improved structure in a multiprocessor program.
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization.
- It makes it easier to use sender-initiated communication, which may have some advantages in performance.

Of course, the desired communication model can be created on top of a hardware model that supports either of these mechanisms. Supporting message passing on top of shared memory is considerably easier: Because messages essentially send data from one memory to another, sending a message can be implemented by doing a copy from one portion of the address space to another. The major difficulties arise from dealing with messages that may be misaligned and of arbitrary length in a memory system that is normally oriented toward transferring aligned blocks of data organized as cache blocks. These difficulties can be overcome either with small performance penalties in software or with essentially no penalties, using a small amount of hardware support.

Supporting shared memory efficiently on top of hardware for message passing is much more difficult. Without explicit hardware support for shared memory, all shared-memory references need to involve the operating system to provide address translation and memory protection, as well as to translate memory references into message sends and receives. Loads and stores usually move small amounts of data, so the high overhead of handling these communications in software severely limits the range of applications for which the performance of software-based shared memory is acceptable. An ongoing area of research is the exploration of when a software-based model is acceptable and whether a software-based mechanism is usable for the highest level of communication in a hierarchically structured system. One possible direction is the use of virtual memory mechanisms to share objects at the page level, a technique called *shared virtual memory;* we discuss this approach in Section 6.10.

In distributed-memory multiprocessors, the memory model and communication mechanisms distinguish the multiprocessors. Originally, distributed-memory multiprocessors were built with message passing, since it was clearly simpler and many designers and researchers did not believe that a shared address space could be built with distributed memory. Shared-memory communication has been supported in virtually every multiprocessor designed since 1995. What hardware communication mechanisms will be supported in the very largest multiprocessors, called *massively parallel processors* (MPPs), which typically have far more than 100 processors, is unclear; shared memory, message passing, and hybrid approaches are all contenders. Despite the symbolic importance of the MPPs, such multiprocessors are a small portion of the market and have little or no influence on the mainstream multiprocessors with tens of processors. We will return to a discussion of the possibilities and trends for MPPs in the concluding remarks and historical perspective at the end of this chapter.

SMPs, which we focus on in Section 6.3, vastly dominate DSM multiprocessors in terms of market size (both units and dollars) and will probably be the architecture of choice for on-chip multiprocessors. For moderate-scale multiprocessors (> 8 processors) long-term technical trends favor distributing memory, which is also likely to be the dominant approach when on-chip SMPs are used as the building blocks in the future. These distributed shared-memory multiprocessors are a natural extension of the centralized multiprocessors that dominate the market, so we discuss these architectures in Section 6.5. In contrast, multicomputers or message-passing multiprocessors build on advances in network technology and are described in Chapter 8. Since the technologies employed were well described in the last chapter, we focus our attention on shared-memory approaches in the rest of this chapter.

Challenges of Parallel Processing

Two important hurdles, both explainable with Amdahl's Law, make parallel processing challenging. The first has to do with the limited parallelism available in programs, and the second arises from the relatively high cost of communications. Limitations in available parallelism make it difficult to achieve good speedups in any parallel processor, as our first example shows.

Example Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

Answer Amdahl's Law is

$$\text{Speedup} = \frac{1}{\dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

For simplicity in this example, assume that the program operates in only two modes: parallel with all processors fully used, which is the enhanced mode, or serial with only one processor in use. With this simplification, the speedup in enhanced mode is simply the number of processors, while the fraction of enhanced mode is the time spent in parallel mode. Substituting into the previous equation:

$$80 = \frac{1}{\dfrac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

Simplifying this equation yields

$$\begin{aligned} 0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) &= 1 \\ 80 - 79.2 \times \text{Fraction}_{\text{parallel}} &= 1 \\ \text{Fraction}_{\text{parallel}} &= \frac{80 - 1}{79.2} \\ \text{Fraction}_{\text{parallel}} &= 0.9975 \end{aligned}$$

Thus to achieve a speedup of 80 with 100 processors, only 0.25% of original computation can be sequential. Of course, to achieve linear speedup (speedup of *n* with *n* processors), the entire program must usually be parallel with no serial portions. (One exception to this is *superlinear speedup* that occurs due to the increased memory and cache available when the processor count is increased. This effect is usually not very large and rarely scales linearly with processor count.) In practice, programs do not just operate in fully parallel or sequential mode, but often use less than the full complement of the processors when running in parallel mode. Exercise 6.1 asks you to extend Amdahl's Law to deal with such a case.

The second major challenge in parallel processing involves the large latency of remote access in a parallel processor. In existing shared-memory multiprocessors, communication of data between processors may cost anywhere from 100 clock cycles to over 1000 clock cycles, depending on the communication mechanism, the type of interconnection network, and the scale of the multiprocessor. Figure 6.3 shows the typical round-trip delays to retrieve a word from a remote memory for several different shared-memory parallel processors.

The effect of long communication delays is clearly substantial. Let's consider a simple example.

Multiprocessor	Year shipped	SMP or NUMA	Maximum processors	Interconnection network	Typical remote memory access time (ns)
Sun Starfire servers	1996	SMP	64	multiple buses	500
SGI Origin 3000	1999	NUMA	512	fat hypercube	500
Cray T3E	1996	NUMA	2048	2-way 3D torus	300
HP V series	1998	SMP	32	8×8 crossbar	1000
Compaq AlphaServer GS	1999	SMP	32	switched buses	400

Figure 6.3 Typical remote access times to retrieve a word from a remote memory in shared-memory multiprocessors.

Example Suppose we have an application running on a 32-processor multiprocessor, which has a 400 ns time to handle reference to a remote memory. For this application, assume that all the references except those involving communication hit in the local memory hierarchy, which is slightly optimistic. Processors are stalled on a remote request, and the processor clock rate is 1 GHz. If the base IPC (assuming that all references hit in the cache) is 2, how much faster is the multiprocessor if there is no communication versus if 0.2% of the instructions involve a remote communication reference?

Answer It is simpler to first calculate the CPI. The effective CPI for the multiprocessor with 0.2% remote references is

$$\begin{aligned}\text{CPI} &= \text{Base CPI} + \text{Remote request rate} \times \text{Remote request cost}\\ &= \frac{1}{\text{Base IPC}} + 0.2\% \times \text{Remote request cost}\\ &= 0.5 + 0.2\% \times \text{Remote request cost}\end{aligned}$$

The remote request cost is

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{400\text{ns}}{1\text{ ns}} = 400 \text{ cycles}$$

Hence we can compute the CPI:

$$\text{CPI} = 0.5 + 0.8 = 1.3$$

The multiprocessor with all local references is 1.3/0.5 = 2.6 times faster. In practice, the performance analysis is much more complex, since some fraction of the noncommunication references will miss in the local hierarchy and the remote access time does not have a single constant value. For example, the cost of a remote reference could be quite a bit worse, since contention caused by many references trying to use the global interconnect can lead to increased delays.

These problems—insufficient parallelism and long-latency remote communication—are the two biggest challenges in using multiprocessors. The problem of inadequate application parallelism must be attacked primarily in software with new algorithms that can have better parallel performance. Reducing the impact of long remote latency can be attacked both by the architecture and by the programmer. For example, we can reduce the frequency of remote accesses with either hardware mechanisms, such as caching shared data, or software mechanisms, such as restructuring the data to make more accesses local. We can try to tolerate the latency by using prefetching or multithreading, which we examined in Chapters 4 and 5.

Much of this chapter focuses on techniques for reducing the impact of long remote communication latency. For example, Sections 6.3 and 6.5 discuss how caching can be used to reduce remote access frequency, while maintaining a coherent view of memory. Section 6.7 discusses synchronization, which, because it inherently involves interprocessor communication, is an additional potential bottleneck. Section 6.8 talks about latency-hiding techniques and memory consistency models for shared memory. Before we wade into these topics, it is helpful to have some understanding of the characteristics of parallel applications, both for better comprehension of the results we show using some of these applications and to gain a better understanding of the challenges in writing efficient parallel programs.

6.2 Characteristics of Application Domains

In earlier chapters, we examined the performance and characteristics of applications with only a small amount of insight into the structure of the applications. For understanding the key elements of uniprocessor performance, such as caches and pipelining, general knowledge of an application is often adequate, although we saw that deeper application knowledge was necessary to exploit higher levels of ILP.

In parallel processing, the additional performance-critical characteristics—such as load balance, synchronization, and sensitivity to memory latency—typically depend on high-level characteristics of the application. These characteristics include factors like how data are distributed, the structure of a parallel algorithm, and the spatial and temporal access patterns to data. Therefore at this point we take the time to examine the three different classes of workloads.

The three different domains of multiprocessor workloads we explore are a commercial workload, consisting of transaction processing, decision support, and web searching; a multiprogrammed workload with operating systems behavior included; and a workload consisting of individual parallel programs from the technical computing domain.