1

Introduction

For over a decade, we have enjoyed explosive growth in the performance and capability of computer systems. The theme of this dramatic success story is the advance of the underlying VLSI technology, which allows clock rates to increase and larger numbers of components to fit on a chip. The plot of this story centers on computer architecture, which translates the raw potential of the technology into greater performance and expanded capability of the computer system. The story's leading character is parallelism. A larger volume of resources means that more operations can be performed at once, in parallel. Parallel computer architecture is about organizing these resources so that they work well together. Computers of all types have harnessed parallelism more and more effectively to gain performance from the raw technology, and the level at which parallelism is exploited continues to rise. Another key character is storage. The data that is operated on at an ever faster rate must be held somewhere in the machine. Thus, the story of parallel processing is deeply intertwined with data locality and communication. The computer architect must sort out these changing relationships to design the various levels of a computer system so as to maximize performance and programmability within the limits imposed by technology and cost at any particular time.

Parallelism is a fascinating perspective from which to understand computer architecture because it applies at all levels of design, it interacts with essentially all other architectural concepts, and it presents a unique dependence on the underlying technology. In particular, the basic issues of locality, bandwidth, latency, and synchronization arise at many levels of the design of parallel computer systems. The trade-offs must be resolved in the context of real application workloads.

Parallel computer architecture, like any other aspect of design, involves elements of form and function. These elements are captured nicely in the following definition (Almasi and Gottlieb 1989):

> A *parallel computer* is a "collection of processing elements that communicate and cooperate to solve large problems fast."

However, this simple definition raises many questions. How large a collection are we talking about? How powerful are the individual processing elements, and can the number be increased in a straightforward manner? How do these elements communicate and cooperate? How is data transmitted between processors, what sort of interconnection is provided, and what operations are available to sequence the actions carried out on different processors? What are the primitive abstractions that

the hardware and software provide to the programmer? And finally, how does it all translate into performance? In answering these questions, we will see that small, moderate, and very large collections of processing elements each have important roles to fill in modern computing. Thus, it is important to understand parallel machine design across the scale, from the small to the very large. Some design issues apply throughout the scale of parallelism; others are most germane to a particular regime, such as within a chip, within a box, or on a very large machine. It is safe to say that parallel machines occupy a rich and diverse design space. This diversity makes the area exciting, but it also means that it is important that we develop a clear framework in which to understand the many design alternatives.

Parallel architecture is itself changing rapidly. Historically, parallel machines have demonstrated innovative organizational structures, often tied to specific programming models, as architects sought to obtain the ultimate in performance out of a given technology. In many cases, radical organizations were justified on the grounds that advances in the base technology would eventually run out of steam. These dire predictions appear to have been overstated, as logic densities and switching speeds have continued to improve and more modest parallelism has been employed at lower levels to sustain continued improvement in processor performance. Nonetheless, application demand for computational performance continues to outpace what individual processors can deliver, and multiprocessor systems occupy an increasingly important place in mainstream computing. What has changed is the novelty of these parallel architectures. Even large-scale parallel machines today are built out of the same basic components as workstations and personal computers. They are subject to the same engineering principles and cost-performance trade-offs. Moreover, to yield the utmost in performance, a parallel machine must extract the full performance potential of its individual components. Thus, an understanding of modern parallel architectures must include an in-depth treatment of engineering trade-offs, not just a descriptive taxonomy of possible machine structures.

Parallel architectures will play an increasingly central role in information processing. This view is based not so much on the assumption that individual processor performance will soon reach a plateau but rather on the estimation that the next level of system design, the multiprocessor level, will become increasingly attractive with increases in chip density. *The goal of this book is to articulate the principles of computer design at the multiprocessor level.* It examines the design issues present for each of the system components—processors, memory systems, and networks—and the relationships between these components. A key aspect is understanding the division of responsibilities between hardware and software in evolving parallel machines. Understanding this division requires familiarity with the requirements that parallel programs place on the machine and the interaction of machine design and the practice of parallel programming.

The process of learning computer architecture is frequently likened to peeling an onion, and this analogy is even more appropriate for parallel computer architecture. At each level of understanding we find a complete whole with many interacting facets, including the structure of the machine, the abstractions it presents, the tech-

nology it rests upon, the software that exercises it, and the models that describe its performance. However, if we dig deeper into any of these facets, we discover another layer of design and a new set of interactions. The holistic, multilevel nature of parallel computer architecture makes the field challenging to learn and challenging to present. Some sense of the layer-by-layer structure is unavoidable.

This introductory chapter presents the "outer skin" of parallel computer architecture. It first outlines the reasons why parallel machine design may become pervasive, from desktop machines to supercomputers. It also examines the technological, architectural, and economic trends that have led to the current state of computer architecture and that provide the basis for anticipating future parallel architectures. Section 1.1 focuses on the forces that have brought about the dramatic advance of processor performance and the restructuring of the entire computing industry around commodity microprocessors. These forces include the insatiable application demand for computing power, the continued improvements in the density and level of integration in VLSI chips, and the utilization of parallelism at higher and higher levels of the architecture.

Next is a quick look at the spectrum of important architectural styles, which give the field such a rich history and contribute to the modern understanding of parallel machines. Within this diversity of design, a common set of design principles and trade-offs arise, driven by the same advances in the underlying technology. These forces are rapidly leading to a convergence in the field, which forms the emphasis of this book. Section 1.2 surveys traditional parallel machines, including shared memory, message passing, data parallel, systolic arrays, and dataflow, and illustrates the different ways that they address common architectural issues. The discussion shows the dependence of parallel architecture on the underlying technology and, more importantly, demonstrates the convergence that has come about with the dominance of microprocessors.

Building on this convergence, Section 1.3 examines the fundamental design issues that cut across parallel machines: what can be named at the machine level as a basis for communication and coordination, what is the latency or time required to perform these operations, and what is the bandwidth or overall rate at which they can be performed? This shift from conceptual structure to performance components provides a framework for quantitative, rather than merely qualitative, study of parallel computer architecture.

With this initial broad understanding of parallel computer architecture in place, the following chapters dig deeper into its technical substance. Chapters 2 and 3 delve into the structure and requirements of parallel programs to provide a basis for understanding the interaction between parallel architecture and applications. Chapter 4 builds a framework for evaluating design decisions in terms of application requirements and performance measurements. Chapters 5 and 6 are a complete study of parallel computer architecture at the limited scale employed widely in commercial multiprocessors—from a few processors to a few tens of processors. The concepts and structures introduced here form the building blocks for more aggressive large-scale designs presented over the final five chapters.

1.1 WHY PARALLEL ARCHITECTURE

Computer architecture, technology, and applications evolve together and have very strong interactions. Parallel computer architecture is no exception. A new dimension is added to the design space—the number of processors—and the design is even more strongly driven by the demand for performance at acceptable cost. Whatever the performance of a single processor at a given time, higher performance can, in principle, be achieved by utilizing many such processors. How much additional performance is gained and at what additional cost depends on a number of factors, which we will explore throughout the book.

To better understand this interaction, let us consider the performance characteristics of the processor building blocks. Figure 1.1[1] illustrates the growth in processor performance over time for several classes of computers (Hennessy and Jouppi 1991). The dashed extensions of the trend lines represent a naive extrapolation of the trends. Although we should be careful in drawing sharp quantitative conclusions from such limited data, the figure suggests several valuable observations.

First, the performance of the highly integrated, single-chip CMOS microprocessor is steadily increasing and is surpassing the larger, more expensive alternatives. Microprocessor performance has been improving at a rate of about 50% per year. The advantages of using small, inexpensive, low-power, mass-produced processors as the building blocks for computer systems with many processors are intuitively clear. However, until recently the performance of the processor best suited to parallel architecture was far behind that of the fastest single-processor system. This is no longer true. Although parallel machines have been built at various scales since the earliest days of computing, the approach is more viable today than ever before because the basic processor building block is better suited to the job.

The second and perhaps more fundamental observation is that change, even dramatic change, is the norm in computer architecture. The continuing process of change has profound implications for the study of computer architecture because we need to understand not only how things are but how they might evolve and why. Change is one of the key challenges in writing this book—and one of the key motivations. Parallel computer architecture has matured to the point where it needs to be studied from a basis of engineering principles and quantitative evaluation of performance and cost. These are rooted in a body of facts, measurements, and designs of real machines. Unfortunately, existing data and designs are necessarily frozen in time

1. The figure is drawn from an influential paper that sought to explain the dramatic changes taking place in the computing industry (Hennessy and Jouppi 1991). The metric of performance is a bit tricky because it reaches across such a range of time and market segment. The study draws data from general-purpose benchmarks, such as the SPEC benchmark, which is widely used to assess performance on technical computing applications (Hennessy and Patterson 1996). After publication, microprocessors continued to track the prediction while mainframes and supercomputers went through tremendous crises and emerged using multiple CMOS microprocessors in their market niche.

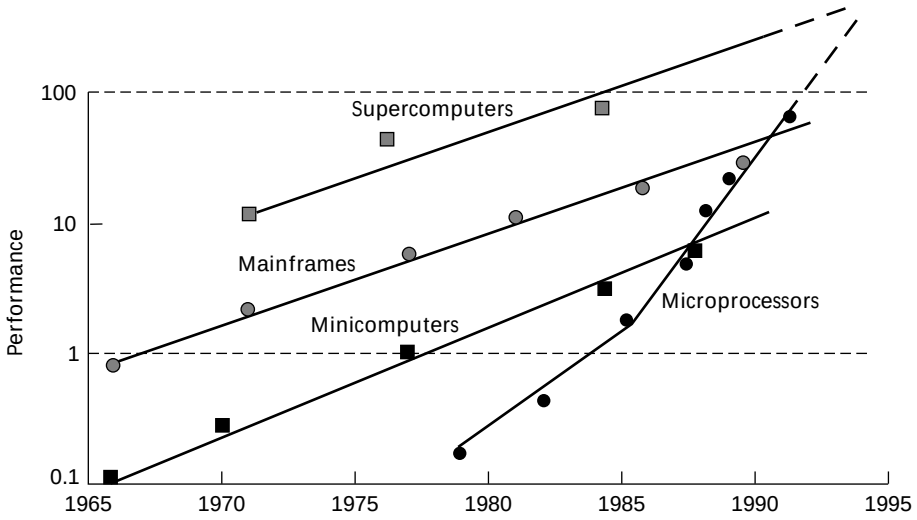


FIGURE 1.1 Performance trends over time of micros, minicomputers, mainframes, and supercomputers. Performance of microprocessors has been increasing at a rate of about 50% per year since the mid-1980s. More traditional mainframe and supercomputer performance has been increasing at a rate of roughly 25% per year. As a result, we are seeing the processor that is best suited to parallel architecture become the performance leader as well. *Source:* Hennessy and Jouppi (1991).

and will become dated as the field progresses. This book presents hard data and examines real machines in the form of a late 1990s technological snapshot in order to retain a clear grounding. However, the methods of evaluation underlying the analysis of concrete design trade-offs transcend the chronological and technological reference point of the book.

The late 1990s happens to be a particularly interesting snapshot because we are in the midst of a dramatic technological realignment as the single-chip microprocessor is poised to dominate every sector of computing and as parallel computing takes hold in many areas of mainstream computing. Of course, the prevalence of change suggests being cautious about extrapolating into the future. The remainder of this section examines more deeply the forces and trends that are giving parallel architectures an increasingly important role throughout the computing field and pushing parallel computing into the mainstream. It looks first at the application demand for increased performance and then at the underlying technological and architectural trends that strive to meet these demands. We see that parallelism is inherently attractive as computers become more highly integrated and that it is being exploited at increasingly high levels of the design. Finally, this section closes with a look at the role of parallelism in the machines at the very high end of the performance spectrum.

1.1.1 Application Trends

The demand for ever greater application performance is a familiar feature of every aspect of computing. Advances in hardware capability enable new application functionality, which grows in significance and places even greater demands on the architecture. This cycle drives the tremendous ongoing design, engineering, and manufacturing effort underlying the sustained exponential performance increase in microprocessor performance. It drives parallel architecture even harder since parallel architecture focuses on the most demanding of these applications. With a 50% annual improvement in processor performance, a parallel machine of a hundred processors can be viewed as providing to applications the computing power that will be widely available 10 years in the future, whereas a thousand processors reflects nearly a 20-year horizon.

Application demand also leads computer vendors to provide a range of models with increasing performance and capacity at progressively increasing cost. The largest volume of machines and the greatest number of users are at the low end, whereas the most demanding applications are served by the high end. One effect of this "platform pyramid" is that the pressure for increased performance is greatest at the high end and is exerted by an important minority of the applications. Prior to the microprocessor era, greater performance was obtained through exotic circuit technologies and machine organizations. Today, to obtain performance significantly greater than the state-of-the-art microprocessor, the primary option is multiple processors, and the most demanding applications are written as parallel programs. Thus, parallel architectures and parallel applications are subject to the most acute demands for greater performance.

A key reference point for both the architect and the application developer is how the use of parallelism improves the performance of the application. We may define the *speedup* on p processors as

$$\text{Speedup}(p \text{ processors}) \equiv \frac{\text{Performance}(p \text{ processors})}{\text{Performance}(1 \text{ processor})} \tag{1.1}$$

For a single, fixed problem, the performance of the machine on the problem is simply the reciprocal of the time to complete the problem, so we have the following important special case:

$$\text{Speedup}_{\textit{fixed problem}}(p \text{ processors}) = \frac{\text{Time}(1 \text{ processor})}{\text{Time}(p \text{ processors})} \tag{1.2}$$

Scientific and Engineering Computing

The direct reliance on increasing levels of performance is well established in a number of endeavors but is perhaps most apparent in the fields of computational science and engineering. Basically, in these fields computers are used to simulate physical phenomena that are impossible or very costly to observe through empirical means.

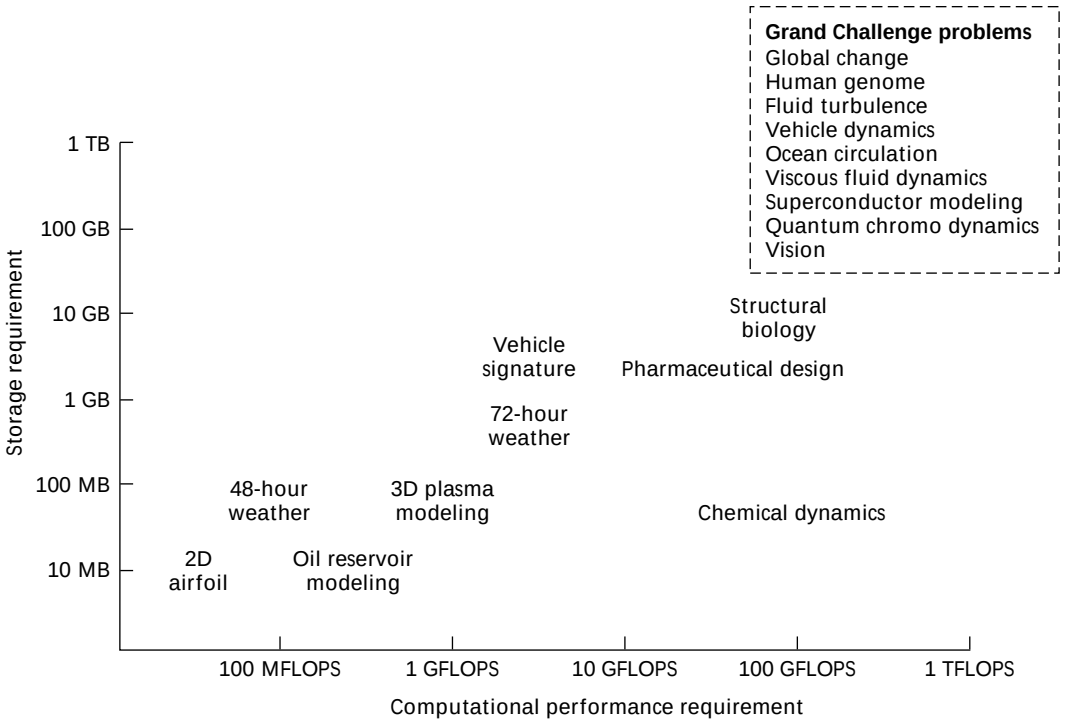


FIGURE 1.2 Grand Challenge application requirements. A collection of important scientific and engineering problems is positioned in a space defined by computational performance and storage capacity. Given the exponential growth rate of performance and capacity, both of these axes map directly to time. In the upper right corner appear some of the Grand Challenge applications identified by the U.S. High Performance Computing and Communications program.

Typical examples include modeling global climate change over long periods, the evolution of galaxies, the atomic structure of materials, the efficiency of combustion with an engine, the flow of air over surfaces of vehicles, the damage due to impacts, and the behavior of microscopic electronic devices. Computational modeling allows in-depth analyses to be performed cheaply on hypothetical designs through computer simulation. A direct correspondence can be drawn between levels of computational performance and the problems that can be studied through simulation. Figure 1.2 summarizes the 1993 findings of the Committee on Physical, Mathematical, and Engineering Sciences of the federal Office of Science and Technology Policy (1993). It indicates the computational rate and storage capacity required to tackle a number of important science and engineering problems. Even with dramatic increases in processor performance, very large parallel architectures are needed to address these problems in the near future. Some years further down the road, new grand challenges will be in view.

Parallel architectures have become the mainstay of scientific computing, including physics, chemistry, material science, biology, astronomy, earth sciences, and others. The engineering application of these tools for modeling physical phenomena is now essential to many industries, including petroleum (reservoir modeling), automotive (crash simulation, drag analysis, combustion efficiency), aeronautics (airflow analysis, engine efficiency, structural mechanics, electromagnetism), pharmaceutical (molecular modeling), and others. In almost all of these applications, there is a large demand for visualization of the results, which is itself a demanding application amenable to parallel computing.

The visualization component has brought the traditional areas of scientific and engineering computing closer to the entertainment industry. In 1995, the first full-length, computer-animated motion picture, *Toy Story,* was produced on a parallel computer system composed of hundreds of Sun workstations. This application was finally possible because the underlying technology and architecture crossed three key thresholds: the decreased cost of computing allowed the rendering to be accomplished within the budget typically associated with a feature film, and the increase in both the performance of individual processors and the scale of parallelism made it possible to complete the task in a reasonable amount of time (several months on several hundred processors). Each science and engineering application has an analogous threshold of computing capacity and cost at which it becomes viable.

Let us take an example from the Grand Challenge program to help understand the strong interaction between applications, architecture, and technology in the context of parallel machines. A 1995 study (Pfeiffer et al. 1995) examined the effectiveness of a wide range of parallel machines on a variety of applications, including a molecular dynamics package, known as AMBER (Assisted Model Building through Energy Refinement). AMBER is widely used to simulate the motion of large biological models such as proteins and DNA, which consist of sequences of residues (amino acids and nucleic acids, respectively) each composed of individual atoms. The code was developed on CRAY vector supercomputers, which employ custom processors, large and expensive SRAM memories (instead of caches), and machine instructions that perform arithmetic or data movement on a sequence, or *vector*, of data values. Figure 1.3 shows the speedup obtained on three versions of this code on a 128-processor microprocessor-based machine—the Intel Paragon, described later. The particular test problem involved the simulation of a protein solvated by water. This test consisted of 99 amino acids and 3,375 water molecules for approximately 11,000 atoms.

The initial parallelization of the code (version 8/94) resulted in good speedup for small configurations but poor speedup on larger configurations. A modest effort to improve the balance of work done by each processor, using techniques discussed in Chapter 2, improved the scaling of the application significantly (version 9/94). An additional effort to optimize communication produced a highly scalable code (version 12/94). This 128-processor version achieved a performance of 406 MFLOPS; the best previously achieved was 145 MFLOPS on a CRAY C90 vector processor. The same application on a more efficient parallel architecture, the CRAY T3D, achieved 891 MFLOPS on 128 processors. This sort of learning curve is quite typical in the

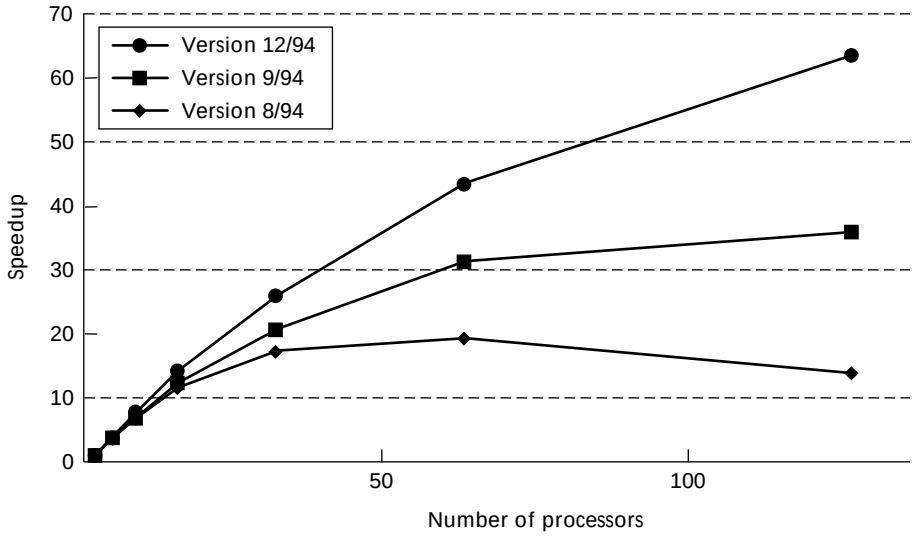


FIGURE 1.3 Speedup on three versions of a parallel program. The parallelization learning curve is illustrated by the speedup obtained on three successive versions of this molecular dynamics code on the Intel Paragon.

parallelization of important applications, as is the interaction between application and architecture. The application writer typically studies the application to understand the demands it places on the available architectures and how to improve its performance on a given set of machines. The architect may study these demands as well in order to understand how to make the machine more effective on a given set of applications. Ideally, the end user of the application enjoys the benefits of both efforts.

The demand for ever increasing performance is a natural consequence of the modeling activity. For example, in electronic CAD there is obviously more to simulate as the number of devices on the chip increases. In addition, the increasing complexity of the design requires that more test vectors be used and, because higher-level functionality is incorporated into the chip, each of these tests must run for a larger number of clock cycles. Furthermore, an increasing level of confidence is required because the cost of fabrication is so great. The cumulative effect is that the computational demand for the design verification of each new generation is increasing at an even faster rate than the performance of the microprocessors themselves.

Commercial Computing

Commercial computing has also come to rely on parallel architectures for its high end. Although the scale of parallelism is typically not as large as in scientific computing, the use of parallelism is even more widespread. Multiprocessors have provided the high end of the commercial computing market since the mid-1960s. In

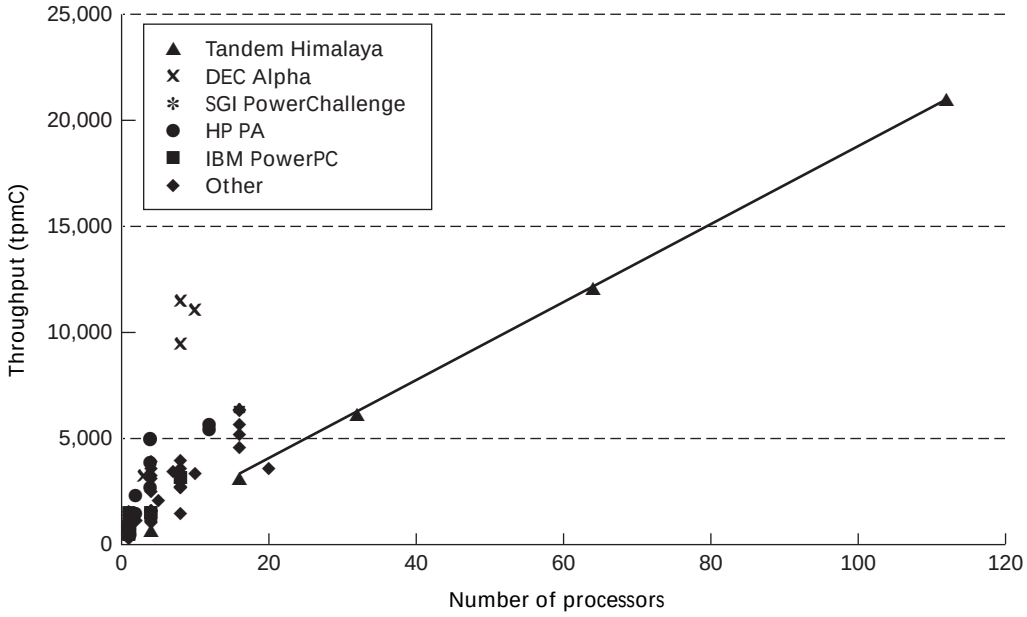


FIGURE 1.4 TPC-C throughput versus number of processors on TPC. The March 1996 TPC report documents the transaction processing performance for a wide range of systems. The figure shows the number of processors employed for all of the high-end systems, highlighting five leading vendor product lines. All of the major database vendors utilize multiple processors for their high-performance options, although the scale of parallelism varies considerably.

this arena, computer system speed and capacity translate directly into the scale of business that can be supported by the system. The relationship between performance and scale of business enterprise is clearly articulated in the on-line transaction processing (OLTP) benchmarks sponsored by the Transaction Processing Performance Council (TPC). These benchmarks rate the performance of a system in terms of its throughput in *transactions per minute* (tpm) on a typical workload. The TPC-C benchmark is an order entry application with a mix of interactive and batch transactions, including realistic features like queued transactions, aborting transactions, and elaborate presentation features (Gray 1991). The benchmark includes explicit scaling criteria to make the problem more realistic: the size of the database and the number of terminals in the system increase as the tpmC (the tpm on TPC-C) rating rises. Thus, a faster system must operate on a larger database and service a larger number of users.

Figure 1.4 shows the tpmC ratings for the collection of systems appearing in one edition of the TPC results (March 1996), with the achieved throughput on the vertical axis and the number of processors employed in the server along the horizontal axis. This data includes a wide range of systems from a variety of hardware and software vendors, a few of which are highlighted here. Since the problem solved in the benchmark run scales with system performance, we cannot simply compare times to

see the effectiveness of parallelism. Instead, we use the throughput of the system as the metric of performance in Equation 1.1. The resulting speedup is illustrated in Example 1.1.

EXAMPLE 1.1 The tpmC for the Tandem Himalaya and IBM PowerPC systems are given in the following table. What is the speedup obtained on each?

tpmC		
Number of Processors	IBM RS6000 PowerPC	Himalaya K10000
1	735	
4	1,438	
8	3,119	
16		3,043
32		6,067
64		12,021
112		20,918

Answer For the IBM system, we may calculate speedup relative to the uniprocessor system; in the Tandem case, we can only calculate speedup relative to a 16-processor system. The IBM machine appears to carry a significant penalty in the parallel database implementation of moving from one to four processors; however, the scaling is very good (superlinear) from four to eight processors. The Tandem system achieves good scaling, although the speedup appears to flatten toward the 100-processor regime. ■

$\text{Speedup}_{\text{tpmC}}$		
Number of Processors	IBM RS6000 PowerPC	Himalaya K10000
1	1	
4	1.96	
8	4.24	
16		1
32		1.99
64		3.95
112		6.87

Several important observations can be drawn from the TPC data. First, the use of parallel architectures is prevalent. Essentially all of the vendors supplying database hardware or software offer multiprocessor systems that provide performance substantially beyond their uniprocessor product. Second, it is not only large-scale parallelism that is important but modest-scale multiprocessor servers with tens of processors and even small-scale multiprocessors with two or four processors.

Finally, even a set of well-documented measurements of a particular class of system at a specific point in time cannot provide a true technological snapshot. Technology evolves rapidly, systems take time to develop and deploy, and real systems have a useful lifetime. Thus, the best systems available from a collection of vendors will be at different points in their life cycle at any time. For example, the DEC Alpha and IBM PowerPC systems in the March 1996 TPC report were much newer than the Tandem Himalaya system. Furthermore, we cannot conclude, for example, that the Tandem system is inherently less efficient as a result of its scalable design. We can, however, conclude that even very large-scale systems must track the technology to retain their advantage.

The transition to parallel programming, including new algorithms or attention to communication and synchronization requirements in existing algorithms, has largely taken place in the high-performance end of computing. The transition is in progress among the much broader base of commercial engineering software. Typically, engineering and commercial applications target more modest-scale multiprocessors, which dominate the server market. In the commercial world, all of the major database vendors support parallel machines for their high-end products. Several major database vendors also offer "shared-nothing" versions for large parallel machines and collections of workstations on a fast network, often called *clusters*. In addition, multiprocessor machines are heavily used to improve throughput on multiprogramming workloads. Even the desktop demonstrates a significant number of concurrent processes, with a host of active windows and daemons. Quite often a single user will have tasks running on many machines within the local area network or will farm tasks out across the network. All of these trends provide a solid application demand for parallel architectures of a variety of scales.

1.1.2 Technology Trends

The importance of parallelism in meeting the application demand for ever greater performance can be brought into sharper focus by looking more closely at the advancements in the underlying technology and architecture. These trends suggest that it may be increasingly difficult to "wait for the single processor to get fast enough" while parallel architectures become more attractive. Moreover, the examination shows that the critical issues in parallel computer architecture are fundamentally similar to those that we wrestle with in "sequential" computers, such as how the resource budget should be divided among functional units that do the work, caches that exploit locality, and wires that provide communication bandwidth.

The primary technological advance is a steady reduction in the basic VLSI feature size. This makes transistors, gates, and circuits faster and smaller, so more fit in the same area. In addition, the useful die size is growing, so there is more area to use. Intuitively, clock rate improves in proportion to the improvement in feature size while the number of transistors grows as the square, or even faster, due to increasing overall die area. Thus, in the long run, the use of many transistors at once (i.e., parallelism) can be expected to contribute more than clock rate to the observed performance improvement of the single-chip building block.

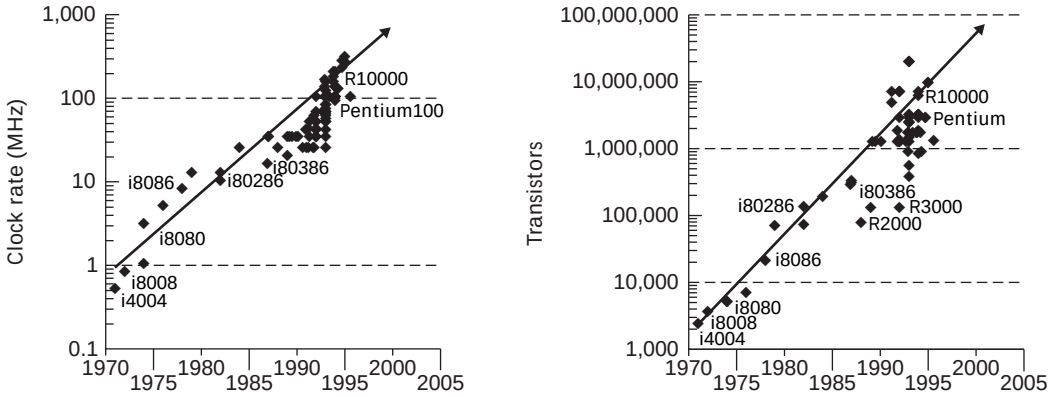


FIGURE 1.5 Improvement in logic density and clock frequency of microprocessors. Improvements in lithographic technique, process technology, circuit design, and datapath design have yielded a sustained improvement in logic density and clock rate.

This intuition is borne out by examination of commercial microprocessors. Figure 1.5 shows the increase in clock frequency and transistor count for several important microprocessor families. Clock rates for the leading microprocessors increase by about 30% per year while the number of transistors increases by about 40% per year. Thus, if we look at the raw computing power of a chip (total transistors switching per second), transistor capacity has contributed an order of magnitude more than clock rate over the past two decades.[2] The performance of microprocessors on standard benchmarks has been increasing at a much greater rate than clock frequency. The most widely used benchmark for measuring workstation performance is the SPEC suite, which includes several realistic integer programs and floating-point programs (SPEC 1995). Integer performance on SPEC has been increasing at about 55% per year and floating-point performance at 75% per year. The LINPACK benchmark (Dongarra 1994) is the most widely used metric of performance on numerical applications. LINPACK floating-point performance has been increasing at more than 80% per year. Thus, processors are getting faster in large part by making more effective use of an ever larger volume of computing resources.

The simplest analysis of these technology trends suggests that the basic single-chip building block will provide increasingly large capacity—in the vicinity of 100 million transistors by the year 2000. This raises the possibility of placing more of the computer system on the chip, including memory and I/O support, or of placing multiple processors on the chip (Gwennap 1994a). The former yields a small and

2. There are many reasons why the transistor count does not increase as the square of the clock rate. One is that much of the area of a processor is consumed by wires, serving to distribute control, data, or clock (i.e., on-chip communication). We will see that the communication issue reappears at every level of parallel computer architecture.

conveniently packaged building block for parallel architectures. The latter brings parallel architecture into the single-chip regime (Gwennap 1994b). Both possibilities are in evidence commercially, with the system-on-a-chip becoming first established in embedded systems, portables, and low-end personal computer products. The use of multiple processors on a chip is becoming established in digital signal processing (Feigel 1994).

The divergence between capacity and speed is much more pronounced in memory technology. From 1980 to 1995, the capacity of a DRAM chip increased a thousand-fold, quadrupling every three years, while the memory cycle time improved by only a factor of two. In the time frame of the 100-million-transistor microprocessor, we anticipate gigabit DRAM chips, but the gap between processor cycle time and memory cycle time will have grown substantially wider. Thus, the memory bandwidth demanded by the processor (bytes per memory cycle) is growing rapidly.

The latency of a memory operation is determined by the access time, which is smaller than the memory cycle time, but still the number of processor cycles per memory access time is large and increasing. To reduce the average latency experienced by the processor and to increase the bandwidth that can be delivered to the processor, we must make more effective use of the levels of the memory hierarchy that lie between the processor and the DRAM memory. Essentially all modern microprocessors provide one or two levels of caches on chip, and most system designs provide an additional level of external cache. A fundamental question as we move into multiprocessor designs is how to organize the collection of caches that lies between the many processors and the many memory modules. For example, one of the immediate benefits of parallel architectures is that the total size of each level of the memory hierarchy can increase with the number of processors without increasing the access time.

Extending these observations to disks, we see a similar divergence. Parallel disk storage systems, such as RAID, are becoming the norm. Large, multilevel caches for files or disk blocks are predominant.

1.1.3 Architectural Trends

Advances in technology determine what is possible; architecture translates the potential of the technology into performance and capability. Fundamentally, the two ways in which a larger volume of resources (e.g., more transistors) improves performance are parallelism and locality. Moreover, these two approaches compete for the same resources. Whenever multiple operations are performed in parallel, the number of cycles required to execute the program is reduced. However, resources are required to support each of the simultaneous activities. Whenever data references are performed close to the processor, the latency of accessing deeper levels of the storage hierarchy is avoided and the number of cycles to execute the program is reduced. However, resources are required to provide this local storage. In general, the best performance is obtained by an intermediate strategy that devotes resources to exploiting a degree of parallelism and a degree of locality. Indeed, we will see throughout the book that parallelism and locality interact in interesting ways in sys-

tems of all scales, from within a chip to across a large parallel machine. In current microprocessors, the die area is divided roughly equally between cache storage, processing, and off-chip interconnect. Larger-scale systems may exhibit a somewhat different split because of differences in cost and performance trade-offs, but the basic issues are the same.

Microprocessor Design Trends

Examining the trends in microprocessor architecture helps build intuition toward the issues we will be dealing with in parallel machines. It also illustrates how fundamental parallelism is to conventional computer architecture and how current architectural trends are leading toward multiprocessor designs. (The discussion of processor design techniques in this book is cursory since many readers are expected to be familiar with those techniques from traditional architecture texts [Hennessy and Patterson 1996] or the many discussions in the trade literature. It does provide a unique perspective on those techniques, however, and will serve to refresh your memory.)

The history of computer architecture has traditionally been divided into four generations identified by the basic logic technology: tubes, transistors, integrated circuits, and VLSI. The entire period covered by the figures in this chapter is lumped into the fourth, or VLSI, generation. Clearly, there has been tremendous architectural advance over this period, but what delineates one era from the next within this generation? The strongest delineation is the kind of parallelism that is exploited as indicated in Figure 1.6.

The period up to about 1986 is dominated by advancements in *bit-level parallelism*, with 4-bit microprocessors replaced by 8-bit, 16-bit, and so on. Doubling the width of the datapath reduces the number of cycles required to perform a full 32-bit operation. Once a 32-bit word size is reached in the mid-1980s, this trend slows, with only partial adoption of 64-bit operation obtained a decade later. Further increases in word width will be driven by demands for improved floating-point representation and a larger address space rather than performance. With address space requirements growing by less than a bit per year, the demand for 128-bit operation appears to be well in the future. The early microprocessor period was able to reap the benefits of the easiest form of parallelism: bit-level parallelism in every operation. The dramatic inflection point in the microprocessor growth curve shown in Figure 1.1 marks the arrival in 1986 of full 32-bit word operation combined with the prevalent use of caches.

The period from the mid-1980s to the mid-1990s is dominated by advancements in *instruction-level parallelism*, performing portions of several machine instructions concurrently. Full-word operation meant that the basic steps in instruction processing (instruction decode, integer arithmetic, and address calculation) could each be performed in a single cycle; with caches, the instruction fetch and data access could also be performed in a single cycle most of the time. The RISC approach demonstrated that, with care in the instruction set design, it was straightforward to pipeline the stages of instruction processing so that an instruction is executed almost every

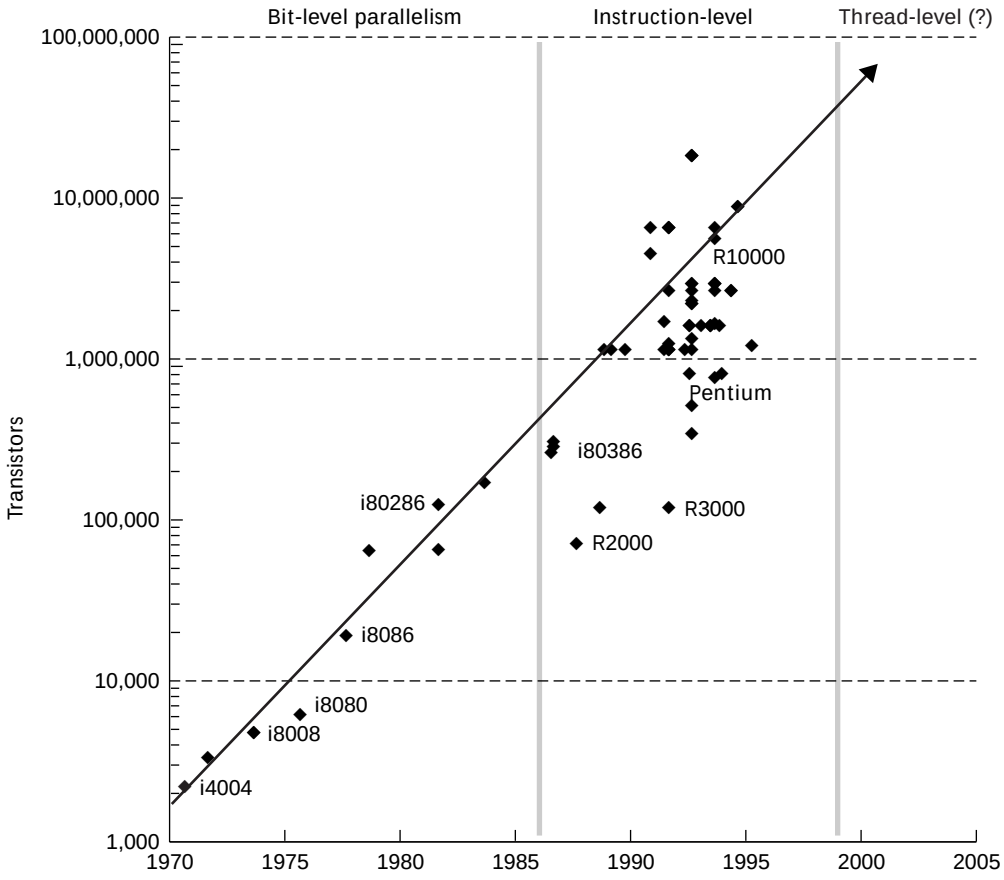


FIGURE 1.6 Number of transistors per processor chip over the last 25 years. The growth essentially follows Moore's Law, which says that the number of transistors doubles every two years. Forecasting from past trends, we can reasonably expect to be designing for a 50- to 100-million-transistor budget at the end of the decade. Also indicated are the epochs of design within the fourth, or VLSI, generation of computer architecture, reflecting the increasing level of parallelism.

cycle, on average. Thus, the parallelism inherent in the steps of instruction processing could be exploited across a small number of instructions. While pipelined instruction processing was not new, it had never before been so well suited to the underlying technology. In addition, advances in compiler technology made instruction pipelines more effective.

The mid-1980s microprocessor-based computers consisted of a small constellation of chips: an integer processing unit, a floating-point unit, a cache controller, and SRAMs for the cache data and tag storage. As chip capacity increased, these components were coalesced into a single chip, which reduced the cost of communicating among them. Thus, a single chip contained separate hardware for integer

arithmetic, memory operations, branch operations, and floating-point operations. In addition to pipelining individual instructions, it became very attractive to fetch multiple instructions at a time and issue them in parallel to distinct function units whenever possible. This form of instruction-level parallelism came to be called *superscalar* execution. It provided a natural way to exploit the ever increasing number of available chip resources. More function units were added, more instructions were fetched at a time, and more instructions could be issued in each clock cycle to the function units.

However, increasing the amount of instruction-level parallelism that the processor can exploit is only worthwhile if the processor can be supplied with instructions and data fast enough to keep it busy. In order to satisfy the increasing instruction and data bandwidth requirement, larger and larger caches were placed on chip with the processor, further consuming the ever increasing number of transistors. With the processor and cache on the same chip, the path between the two could be made very wide to satisfy the bandwidth requirement of multiple instruction and data accesses per cycle. However, as more instructions are issued each cycle, the performance impact of each control transfer and each cache miss becomes more significant. A control transfer may have to wait for the depth, or *latency*, of the processor pipeline until a particular instruction reaches the end of the pipeline and determines which instruction to execute next. Similarly, instructions that use a value loaded from memory may cause the processor to wait for the latency of a cache miss.

Processor designs in the 1990s deploy a variety of complex instruction processing mechanisms in an effort to reduce the performance degradation resulting from latency in "wide-issue" superscalar processors. Sophisticated branch prediction techniques are used to avoid pipeline latency by guessing the direction of control flow before branches are actually resolved. Larger, more sophisticated caches are used to *avoid* the latency of cache misses. Instructions are scheduled dynamically and allowed to complete out of order so if one instruction encounters a miss, other instructions can proceed ahead of it as long as they do not depend on the result of the instruction. A larger window of instructions that are waiting to issue is maintained within the processor and whenever an instruction produces a new result, several waiting instructions may be issued to the function units. These complex mechanisms allow the processor to *tolerate* the latency of a cache miss or pipeline dependence when it does occur. However, each of these mechanisms places a heavy demand on chip resources and carries a very heavy design cost.

Given the expected increases in chip density, the natural question to ask is how far will instruction-level parallelism go within a single thread of control? At what point will the emphasis shift to supporting the higher levels of parallelism available as multiple processes or multiple threads of control within a process, that is, *thread-level parallelism?* Several research studies have sought to answer the first part of the question, either through simulation of aggressive machine designs (Chang et al. 1991; Horst, Harris, and Jardine 1990; Lee, Kwok, and Briggs 1991; Melvin and Patt 1991) or through analysis of the inherent properties of programs (Butler et al. 1991; Jouppi and Wall 1989; Johnson 1991; Smith, Johnson, and Horowitz 1989; Wall 1991). The most complete treatment appears in Johnson's book devoted to the topic

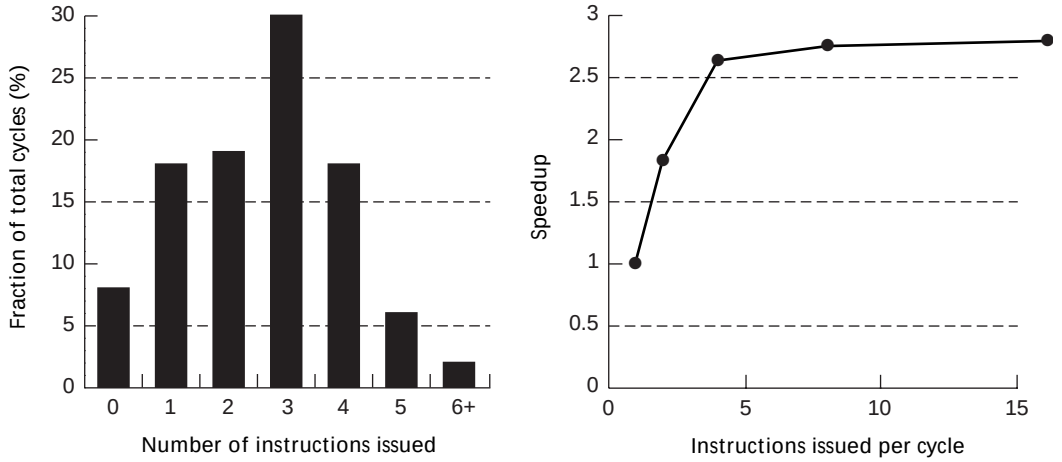


FIGURE 1.7 Distribution of potential instruction-level parallelism and estimated speedup under ideal superscalar execution. The figure shows the distribution of available instruction-level parallelism and maximum potential speedup under idealized superscalar execution, including unbounded processing resources and perfect branch prediction. Data is an average of that presented for several benchmarks by Johnson (1991).

(1991). Simulation of aggressive machine designs generally shows that two-way superscalar, that is, issuing two instructions per cycle, is very profitable and four-way offers substantial additional benefit, but wider issue widths (e.g., eight-way superscalar) provide little additional gain. The design complexity increases dramatically because control transfers occur roughly once in five instructions, on average.

To estimate the maximum potential speedup that can be obtained by issuing multiple instructions per cycle, the execution trace of a program is simulated on an ideal machine with unlimited instruction fetch bandwidth, as many function units as the program can use, and perfect branch prediction. (The latter is easy, since the trace correctly follows each branch.) These generous machine assumptions ensure that no instruction is held up because a function unit is busy or because the instruction is beyond the lookahead capability of the processor. Furthermore, to ensure that no instruction is delayed because it updates a location that is used by logically previous instructions, storage resource dependences are removed by a technique called *renaming*. Each update to a register or memory location is treated as introducing a new "name," and subsequent uses of the value in the execution trace refer to the new name. In this way, the execution order of the program is constrained only by essential data dependences; each instruction is executed as soon as its operands are available. Figure 1.7 summarizes the result of this ideal machine analysis based on data presented by Johnson (1991). The histogram on the left shows the fraction of cycles in which no instruction could issue, only one instruction could issue, and so on. Johnson's ideal machine retains realistic function unit latencies, including cache

misses, which accounts for the zero-issue cycles. (Other studies ignore cache effects or ignore pipeline latencies and thereby obtain more optimistic estimates.) We see that, even with infinite machine resources, perfect branch prediction, and ideal renaming, no more than four instructions issue in a cycle 90% of the time. Based on this distribution, we can estimate the speedup obtained at various issue widths, as shown in the right portion of the figure. Recent work (Lam and Wilson 1992; Sohi, Breach, and Vijaykumar 1995) provides empirical evidence that to obtain significantly larger amounts of parallelism, multiple threads of control must be pursued simultaneously. Barring some unforeseen breakthrough in instruction-level parallelism, the leap to the next level of useful parallelism—multiple concurrent threads—is increasingly compelling as chips increase in capacity.

System Design Trends

The trend toward thread- or process-level parallelism has been strong at the computer system level for some time. Computers containing multiple state-of-the-art microprocessors sharing a common memory became prevalent in the mid-1980s, when the 32-bit microprocessor was first introduced (Bell 1985). As indicated by Figure 1.8, which shows the number of processors available in commercial multiprocessors over time, this bus-based shared memory multiprocessor approach has maintained a substantial multiplier to the increasing performance of the individual processors. Almost every commercial microprocessor introduced since the mid-1980s provides hardware support for multiprocessor configurations, as discussed in Chapter 5. Multiprocessors dominate the server and enterprise (or mainframe) markets and have migrated to the desktop.

The early multi-microprocessor systems were introduced by small companies competing for a share of the minicomputer market, including Synapse (Nestle and Inselberg 1985), Encore (Schanin 1986), Flex (Matelan 1985), Sequent (Rodgers 1985), and Myrias (Savage 1985). They combined 10 to 20 microprocessors to deliver competitive throughput on time-sharing loads. With the introduction of the 32-bit Intel i80386 as the base processor, these systems obtained substantial commercial success, especially in transaction processing. However, the rapid performance advance of RISC microprocessors, exploiting instruction-level parallelism, sapped the CISC multiprocessor momentum in the late 1980s and all but eliminated the minicomputer. Shortly thereafter, several large companies began producing RISC multiprocessor systems, especially as servers and mainframe replacements. These designs highlight the critical role of bandwidth. In most of these multiprocessor designs, all the processors plug into a common bus. Since a bus has a fixed bandwidth, as the processors become faster, a smaller number can be supported by the bus. The early 1990s brought a dramatic advance in the shared memory bus technology, including faster electrical signaling, wider datapaths, pipelined protocols, and multiple paths. Each of these provided greater bandwidth, growing with time and design experience, as indicated in Figure 1.9. This increase in bandwidth allowed the multiprocessor designs to ramp back up to the 10-to-20 range and beyond while tracking the microprocessor advances (Alexander et al. 1994; Cekleov et al. 1993;

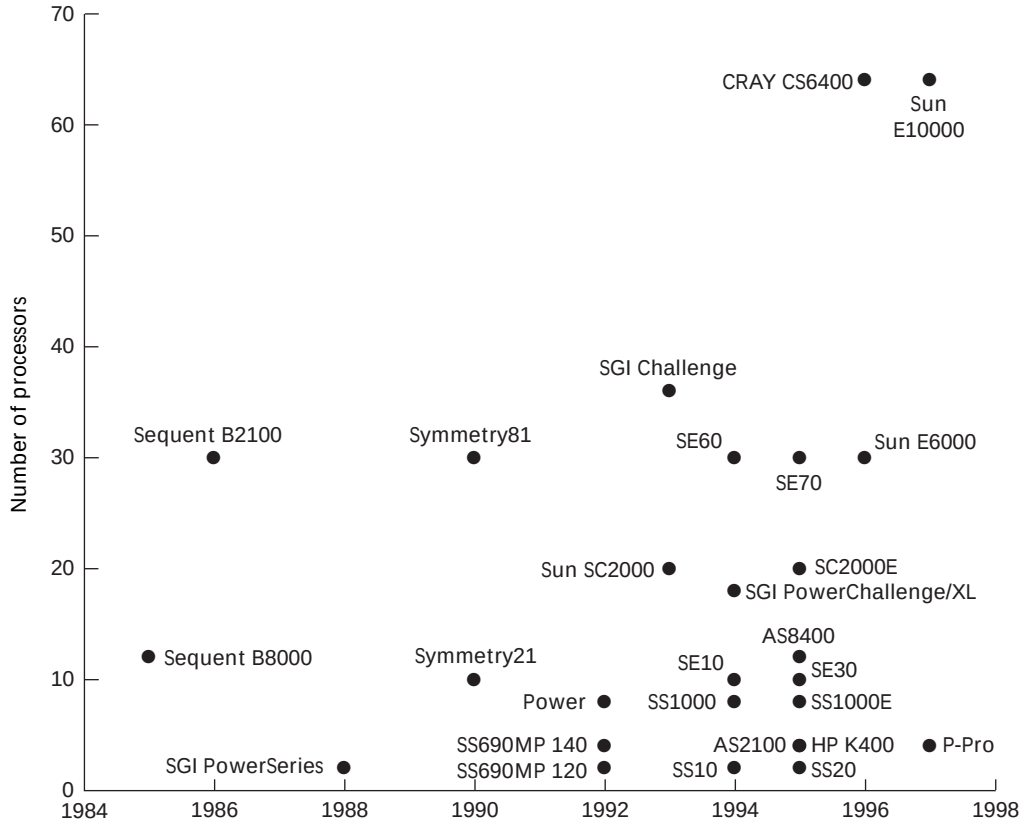


FIGURE 1.8 Number of processors in fully configured commercial bus-based shared memory multiprocessors. After an initial era of 10- to 20-way shared memory processors based on slow CISC microprocessors, companies such as Sun, HP, DEC, SGI, IBM, and CRI began producing sizable RISC-based SMPs, as did commercial vendors not shown here, including NCR/ATT, Tandem, and Pyramid.

Fenwick et al. 1995; Frank, Burkhardt, and Rothnie 1993; Galles and Williams 1993; Godiwala and Maskas 1995).

The picture in the mid-1990s is very interesting. Not only has the bus-based shared memory multiprocessor approach become ubiquitous in the industry, it is present at a wide range of scale. Desktop systems and small servers commonly support two to four processors, larger servers support tens, and large commercial systems are moving toward one hundred. Indications are that this trend will continue. As an illustration of the shift in emphasis, in 1994 Intel defined a standard approach to the design of multiprocessor PC systems around its Pentium microprocessor (Slater 1994). The follow-on Pentium Pro microprocessor allowed four-processor configurations to be constructed by wiring the chips together without even any glue logic; bus drivers, arbitration, and so on are in the microprocessor. This development is expected to make small-scale multiprocessors a true commodity. Addition-

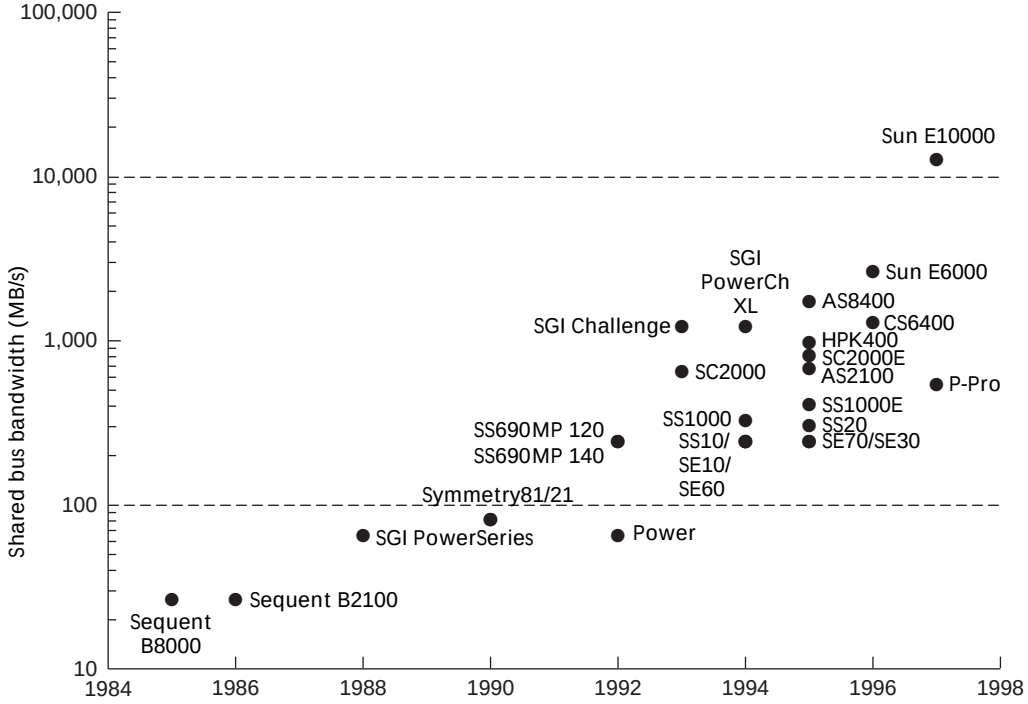


FIGURE 1.9 Bandwidth of the shared memory bus in commercial multiprocessors. After slow growth for several years, a new era of memory bus design began in 1991, which supported the use of substantial numbers of very fast microprocessors.

ally, a shift in the industry business model has been noted, where multiprocessors are being pushed by software vendors, especially database companies, rather than just by the hardware vendors. Combining these trends with the technology trends, it appears that the question is when, not if, multiple processors per chip will become prevalent.

1.1.4 Supercomputers

We have looked at the forces driving the development of parallel architecture in the general market. A second, confluent set of forces comes from the quest to achieve absolute maximum performance, known as *supercomputing*. Although commercial and information processing applications are increasingly becoming important drivers of the high end, scientific computing has historically been a kind of proving ground for innovative architecture. In the mid-1960s, this included pipelined instruction processing and dynamic instruction scheduling, which are commonplace in microprocessors today. Starting in the mid-1970s, supercomputing was dominated by *vector processors*, which perform operations on sequences of data

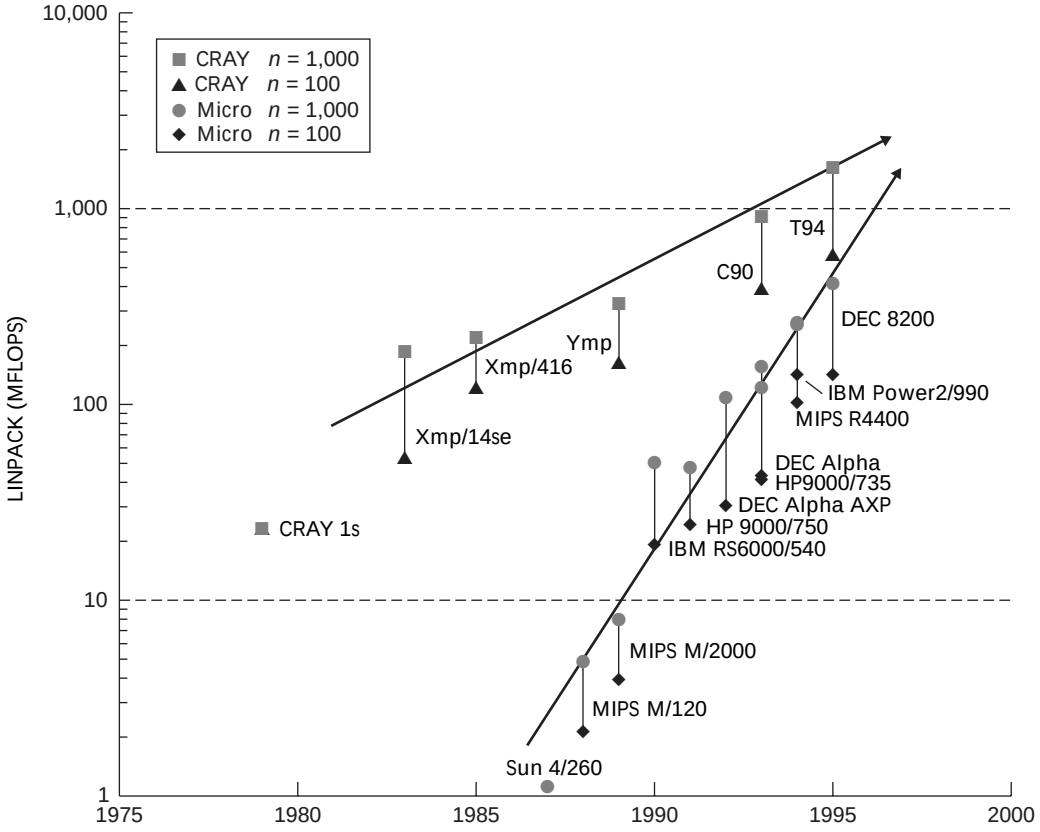


FIGURE 1.10 Uniprocessor performance of supercomputers and microprocessor-based systems on the LINPACK benchmark. Performance in MFLOPS for a single processor on solving dense linear equations is shown for the leading CRAY vector supercomputer and the fastest workstations on a 100×100 and $1{,}000 \times 1{,}000$ matrix.

elements; that is, a vector rather than individual scalar data. Vector operations permit more parallelism to be obtained within a single thread of control. In addition, these vector supercomputers were implemented in very fast, expensive, high-power circuit technologies.

Dense linear algebra is an important component of scientific computing and the specific emphasis of the LINPACK benchmark. Although this benchmark evaluates a narrow aspect of system performance, it is one of the few measurements available for a very wide class of machines over a long period of time. Figure 1.10 shows the LINPACK performance trend for one processor of the leading CRAY vector supercomputers (August et al. 1989; Russel 1978) compared with that of the fastest contemporary microprocessor-based workstations and servers. For each system two

data points are provided. The lower one is the performance obtained on a 100×100 matrix and the higher one on a $1{,}000 \times 1{,}000$ matrix. Within the vector processing approach, the single-processor performance improvement is dominated by modest improvements in cycle time and more substantial increases in the vector memory bandwidth. In the microprocessor systems, we see the combined effect of increasing clock rate, using on-chip pipelined floating-point units, increasing on-chip cache size, increasing off-chip second-level cache size, and increasing use of instruction-level parallelism. The gap in uniprocessor performance is rapidly closing.

Multiprocessor architectures are adopted by both the vector processor and microprocessor designs, but the scale is quite different. The CRAY Xmp first provided two and then four processors, the Ymp eight, the C90 sixteen, and the T94 thirty-two. The microprocessor-based supercomputers initially provided about 100 processors, increasing to roughly 1,000 from 1990 on. These aggregations of processors, known as *massively parallel processors* (MPPs), have tracked the microprocessor advance, with typically a lag of one to two years behind the leading microprocessor-based workstation or personal computer. As shown in Figure 1.11, the large number of slightly slower microprocessors has proved dominant for the LINPACK benchmark. (Note the change of scale from MFLOPS in Figure 1.10 to GFLOPS here.) The performance advantage of the MPP systems over traditional vector supercomputers is less substantial on more complete applications (Bailey et al. 1994) owing to the relative immaturity of the programming languages, compilers, and algorithms; however, the trend toward MPPs is still very pronounced. The importance of this trend was apparent enough in 1993 that CRAY Research announced its T3D, based on the DEC Alpha microprocessor.

Recently, the LINPACK benchmark has been used to rank the fastest computer systems in the world. Figure 1.12 shows the number of multiprocessor parallel vector processors (PVPs), MPPs, and bus-based shared memory multiprocessors (SMPs) appearing in the list of the top 500 systems. The latter two are both microprocessor based, and the trend is clear.

1.1.5 Summary

In examining current trends from a variety of perspectives—economics, technology, architecture, and application demand—we see that parallel architecture is increasingly attractive and increasingly central. The quest for performance is so keen that parallelism is being exploited at many different levels and at various points in the computer design space. Instruction-level parallelism is exploited in all modern high-performance processors. Essentially, all machines beyond the desktop are multiprocessors, including servers, mainframes, and supercomputers. The very high end of the performance curve is dominated by massively parallel processors. The use of large-scale parallelism in applications is broadening. The focus of this book is the multiprocessor level of parallelism. We study the design principles embodied in parallel machines from the modest scale to the very large, so that we may understand the spectrum of viable parallel architectures that can be built from well-proven components.

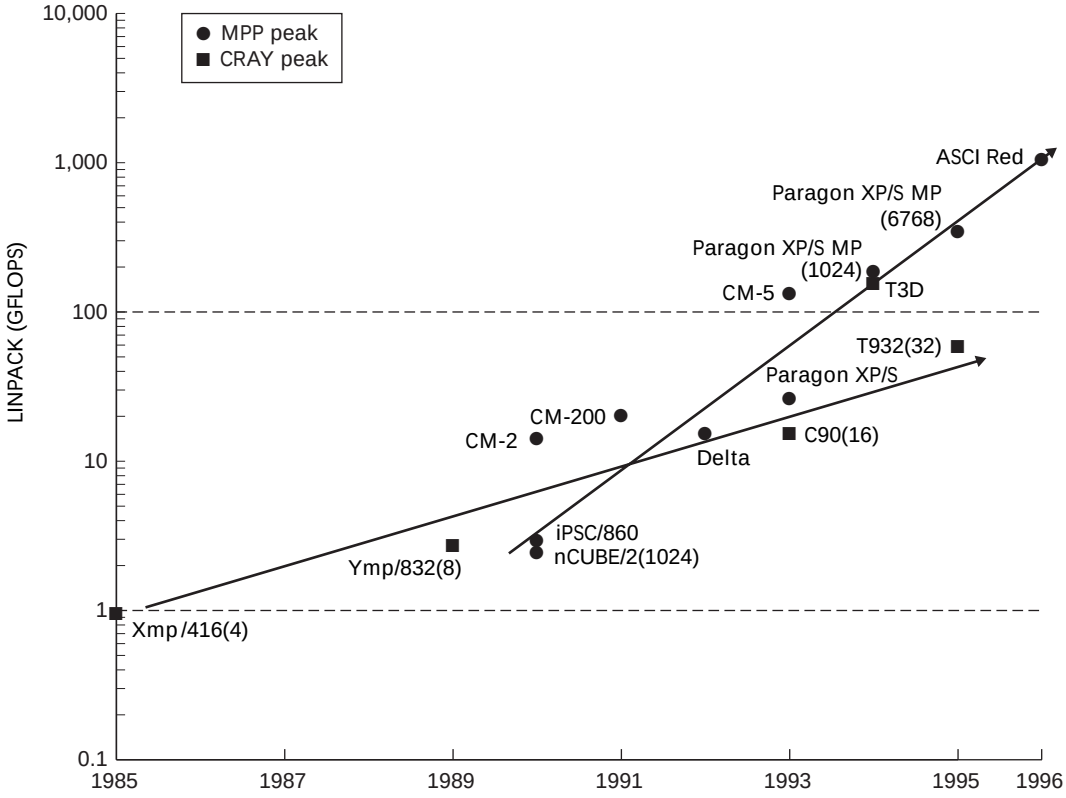


FIGURE 1.11 Performance of supercomputers and MPPs on the LINPACK peak performance benchmark. Peak performance in GFLOPS for solving dense linear equations is shown for the leading CRAY multiprocessor vector supercomputer and the fastest MPP systems. Note the change in scale from Figure 1.10 (MFLOPS to GFLOPS).

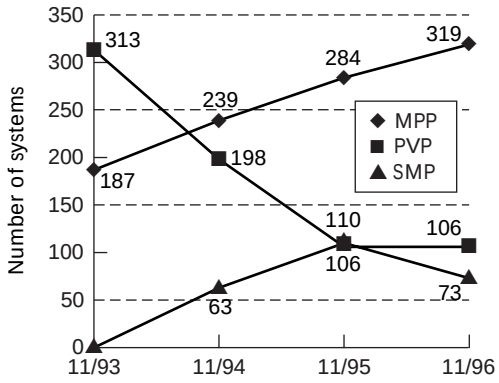


FIGURE 1.12 Types of systems used in the 500 fastest computer systems in the world. Parallel vector processors (PVPs) have given way to microprocessor-based massively parallel processors (MPPs) and bus-based symmetric shared memory multiprocessors (SMPs) at the high end of computing.

This discussion of the trends toward parallel computers has been primarily from the processor perspective, but you may arrive at the same conclusion from the memory system perspective. Consider briefly the design of a memory system to support a very large amount of data, that is, the data set of *large* problems. One of the few physical laws of computer architecture is that fast memories are small, large memories are slow. This occurrence is due to many factors, including the increased address decode time, the delays on increasingly long bit lines, the small drive of increasingly dense storage cells, and the selector delays. The result is that memory systems are constructed as a hierarchy of increasingly larger and slower memories: on average, a large hierarchical memory is fast, as long as the references exhibit good locality. The other trick we can play to cheat the laws of physics and obtain fast access on a very large data set is to use multiple processors and have the different processors access independent smaller memories. Of course, physics is not easily fooled. We pay the cost when a processor accesses nonlocal data, which we call *communication*, and when we need to orchestrate the actions of the many processors (i.e., in synchronization operations).

1.2 CONVERGENCE OF PARALLEL ARCHITECTURES

Historically, parallel machines have developed within several distinct architectural camps, and most texts on the subject are organized around a taxonomy of these designs. However, in looking at the evolution of parallel architecture, it is clear that the designs are strongly influenced by the same technological forces and similar application requirements. It is not surprising therefore that a great deal of convergence has occurred in the field. The goal of this section is to construct a framework for understanding the entire spectrum of parallel computer architectures and to build intuition as to the nature of the convergence. Along the way comes a quick overview of the evolution of parallel machines, starting from the traditional camps and moving toward the point of convergence.

1.2.1 Communication Architecture

Given that a parallel computer is "a collection of processing elements that communicate and cooperate to solve large problems fast" (Almasi and Gottlieb 1989), we may reasonably view parallel architecture as the extension of conventional computer architecture to address issues of communication and cooperation among processing elements. In essence, parallel architecture extends the usual concepts of a computer architecture with a communication architecture. Computer architecture has two distinct facets. One is the definition of critical abstractions, especially the hardware/software boundary and the user/system boundary. The architecture specifies the set of operations at the boundary and the data types that these operate on. The other facet is the organizational structure that realizes these abstractions to deliver high performance in a cost-effective manner. A *communication architecture* has these two

facets as well. It defines the basic communication and synchronization operations, and it addresses the organizational structures that realize these operations.

The framework for understanding communication in a parallel machine is illustrated in Figure 1.13. The top layer is the programming model, which is the conceptualization of the machine that the programmer uses in coding applications. Each programming model specifies how parts of the program running in parallel communicate information to one another and what synchronization operations are available to coordinate their activities. Applications are written in a programming model. In the simplest case, the model consists of multiprogramming a large number of independent sequential programs; no communication or cooperation takes place at the programming level. The more interesting cases include true parallel programming models, such as shared address space, message passing, and data parallel programming. We can describe these models intuitively as follows:

- *Shared address* programming is like using a bulletin board, where you can communicate with one or many colleagues by posting information at known, shared locations. Individual activities can be orchestrated by taking note of who is doing what task.
- *Message passing* is akin to telephone calls or letters, which convey information from a specific sender to a specific receiver. There is a well-defined event when the information is sent or received, and these events are the basis for orchestrating individual activities. However, no shared location is accessible to all.
- *Data parallel* processing is a more regimented form of cooperation, where several agents perform an action on separate elements of a data set simultaneously and then exchange information globally before continuing en masse. The global reorganization of data may be accomplished through accesses to shared addresses or messages since the programming model only defines the overall effect of the parallel steps.

A more precise definition of these programming models will be developed later in the text; at this stage, it is most important to understand the layers of abstraction.

A programming model is realized in terms of the user-level communication primitives of the system, referred to here as the *communication abstraction*. Typically, the programming model is embodied in a parallel language or programming environment, so a mapping exists from the generic language constructs to the specific primitives of the system. These user-level primitives may be provided directly by the hardware, by the operating system, or by machine-specific user software that maps the communication abstractions to the actual hardware primitives. The distance between the lines in Figure 1.13 is intended to indicate that the mapping from one layer to the next may be very simple or very involved. For example, access to a shared location is realized directly by load and store instructions on a machine in which all processors use the same physical memory; however, passing a message on such a machine may involve a library or system call to write the message into a buffer area or to read it out.

The communication architecture defines the set of communication operations available to the user software, the format of these operations, and the data types they

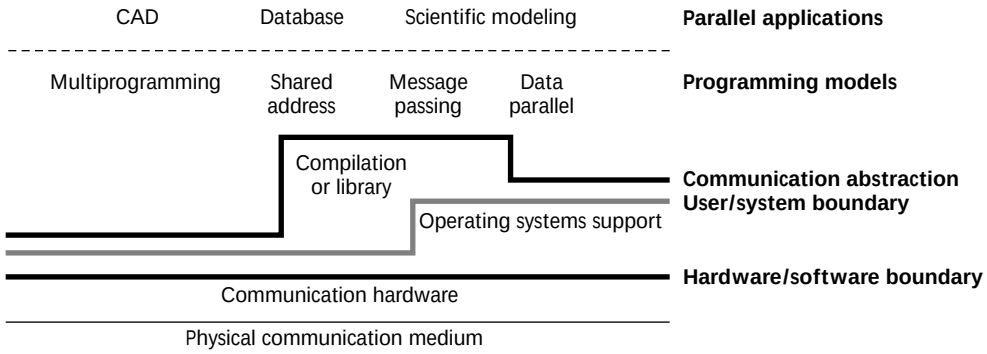


FIGURE 1.13 Layers of abstraction in parallel computer architecture. Critical layers of abstractions lie between the application program and the actual hardware. The application is written for a programming model, which dictates how pieces of the program share information and coordinate their activities. The specific operations providing communication and synchronization form the communication abstraction, which is the boundary between the user program and the system implementation. This abstraction is realized through compiler or library support using the primitives available from the hardware or from the operating system, which uses privileged hardware primitives. The communication hardware is organized to provide these operations efficiently on the physical wires connecting the machine together.

operate on, much as an instruction set architecture does for a processor. Note that even in conventional instruction sets, some operations may be realized by a combination of hardware and software, such as a load instruction that relies on operating system intervention in the case of a page fault. The communication architecture also extends the computer organization with the hardware structures that support communication.

As with conventional computer architecture, a great deal of debate has gone on over the years about what should be incorporated into each layer of abstraction in parallel architecture and how large the gap between the layers should be. This debate has been fueled by differing assumptions about the underlying technology and more qualitative assessments of "ease of programming." The hardware/software boundary in Figure 1.13 is depicted as flat to indicate that the available hardware primitives in different designs is more or less of uniform complexity. Indeed, this is becoming more the case as the field matures. In most early designs, the physical hardware organization was strongly oriented toward a particular programming model; that is, the communication abstraction supported by the hardware was essentially identical to the programming model. This "high-level" parallel architecture approach resulted in tremendous diversity in the hardware organizations. However, as the programming models have become better understood and implementation techniques have matured, compilers and run-time libraries have grown to provide an important bridge between the programming model and the underlying hardware. Simultaneously, the technological trends discussed in Section 1.1.2 have exerted a strong influence, regardless of the programming model. The result has

been a convergence in the organizational structure with relatively simple, general-purpose communication primitives.

Sections 1.2.2–1.2.6 survey the most widely used programming models and the corresponding styles of machine design in past and current parallel machines. With the historical orientation to a particular programming model, it was common to lump the programming model, the communication abstraction, and the machine organization together as "the architecture," for example, shared memory architecture, message-passing architecture, and so on. This approach is less appropriate today since a large commonality exists across parallel machines and since many machines support several programming models. It is important to see how this convergence has come about, so these sections begin from the traditional perspective, looking at machine designs associated with particular programming models and explaining their intended roles and the technological opportunities that influenced their design. The goal of the survey is not to develop a taxonomy of parallel machines per se but to identify a set of core concepts that form the basis for assessing design trade-offs across the entire spectrum of potential designs today and in the future. It also demonstrates the influence that the dominant technological direction established by microprocessor and DRAM technologies has had on parallel machine design, which makes a common treatment of the fundamental design issues natural or even imperative. Specifically, shared address, message-passing, data parallel, dataflow, and systolic approaches are presented. In each case, the abstraction embodied in the programming model is explained, and the reasons for the particular style of design, as well as the intended scale and application, are presented. The technological motivations for the approach are also examined, as well as how they have changed over time. These changes are reflected in the machine organization, which determines what is fast and what is slow. The performance characteristics ripple up to influence aspects of the programming model. The outcome of this brief survey is a clear organizational convergence, which is captured in a generic parallel machine in Section 1.2.7.

1.2.2 Shared Address Space

One of the most important classes of parallel machines is *shared memory multiprocessors*. The key property of this class is that communication occurs implicitly as a result of conventional memory access instructions (i.e., loads and stores). This class has a long history, dating at least to precursors of mainframes in the early 1960s,[3] and today it has a role in almost every segment of the computer industry. Shared memory multiprocessors serve to provide better throughput on multiprogramming workloads, as well as to support parallel programs. Thus, they are naturally found across a wide range of scale, from a few processors to perhaps hundreds. This sec-

3. Some say that BINAC was the first multiprocessor, but it was intended to improve reliability. The two processors checked each other at every instruction. They seldom agreed, so people eventually turned one of them off.

tion examines the communication architecture of shared memory machines and the key organizational issues for small-scale designs and large configurations.

The primary programming model for these machines is essentially that of time-sharing on a single processor, except that real parallelism replaces interleaving in time. Formally, a *process* is a virtual address space and one or more threads of control. Processes can be configured so that portions of their address space are shared, that is, are mapped to a common physical location, as suggested by Figure 1.14. (Multiple threads within a process, by definition, share portions of the address space.) Cooperation and coordination among threads is accomplished by reading and writing shared variables and pointers referring to shared addresses. *Writes to a logically shared address by one thread are visible to reads of the other threads.* The communication architecture employs the conventional memory operations to provide communication through shared addresses as well as special atomic operations for synchronization. Even completely independent processes typically share the kernel portion of the address space, although this is only accessed by operating system code. Nonetheless, the shared address space model is utilized within the operating system to coordinate the execution of the processes.

Although shared memory can be used for communication among arbitrary collections of processes, most parallel programs are quite structured in their use of the virtual address space. They typically have a common code image, private segments for the stack and other private data, and shared segments that are in the same region of the virtual address space of each process or thread of the program. This simple structure implies that the private variables in the program are present in each process and that shared variables have the same address and meaning in each thread. Often, straightforward parallelization strategies are employed. For example, each process may perform a subset of the iterations of a common parallel loop or, more generally, processes may operate as a pool of workers obtaining work from a shared queue. Chapter 2 discusses the structure of parallel programs more deeply. Here we look at the basic evolution and development of this important architectural approach.

The communication hardware for shared memory multiprocessors is a natural extension of the memory system found in most computers. Essentially all computer systems allow a processor and a set of I/O controllers to access a collection of memory modules through some kind of hardware interconnect, as illustrated in Figure 1.15. The memory capacity is increased simply by adding memory modules. Additional capacity may or may not increase the available memory bandwidth, depending on the specific system organization. I/O capacity is increased by adding devices to I/O controllers or by inserting additional I/O controllers. There are two possible ways to increase the processing capacity: wait for a faster processor to become available or add more processors. On a time-sharing workload, increasing processing capacity should increase the throughput of the system. With more processors, more processes can run at once and throughput is increased. If a single application is programmed to make use of multiple threads, more processors should speed up the application. The hardware primitives are essentially one to one with the communication abstraction, and these operations are available in the programming model.

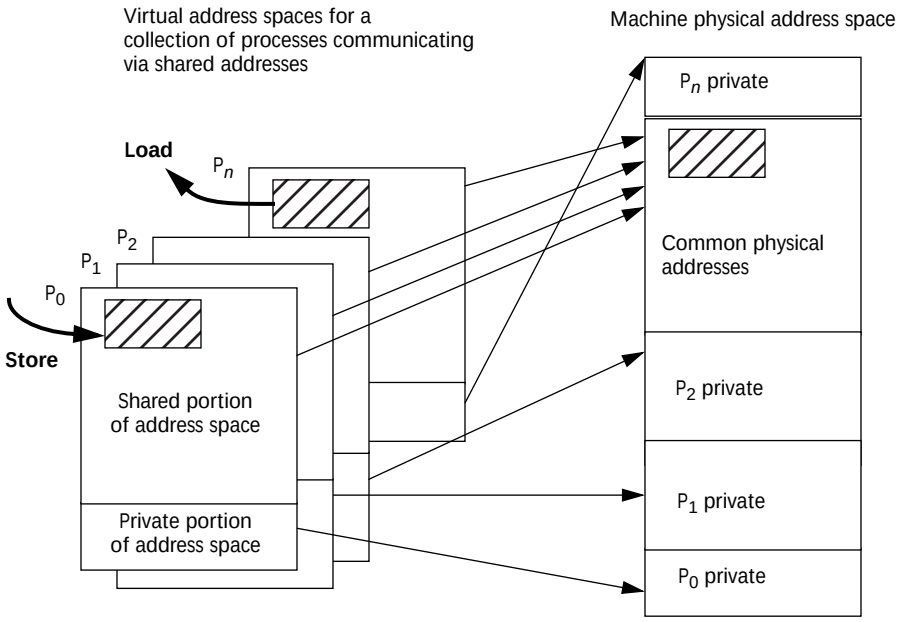


FIGURE 1.14 Typical memory model for shared memory parallel programs. Collections of processes have a common region of physical addresses mapped into their virtual address space, in addition to the private region, which typically contains the stack and private data.

Within the general framework of Figure 1.15, a great deal of evolution of shared memory machines has taken place as the underlying technology has advanced. The early machines were "high-end" mainframe configurations (Lonergan and King 1961; Padegs 1981). On the technology side, memory in early mainframes was slow compared to the processor, so it was necessary to interleave data across several memory banks to obtain adequate bandwidth for even a single processor; this required an interconnect between the processor and each of the banks. On the application side, these systems were primarily designed for throughput on a large number of jobs. Thus, to meet the I/O demands of a workload, several I/O channels and devices were attached. The I/O channels also required direct access to each of the memory banks. Therefore, these systems were typically organized with a *crossbar switch* connecting the CPU and several I/O channels to several memory banks, as indicated by Figure 1.16a. Adding processors was primarily a matter of expanding the switch; the hardware structure to access a memory location from a port on the processor and I/O side of the switch was unchanged. The size and cost of the processor limited these early systems to a small number of processors, but as the hardware density and cost improved, larger systems could be contemplated. The cost of scaling the crossbar became the limiting factor, and in many cases it was replaced by a *multistage interconnect*, suggested by Figure 1.16b, for which the cost increases more slowly with

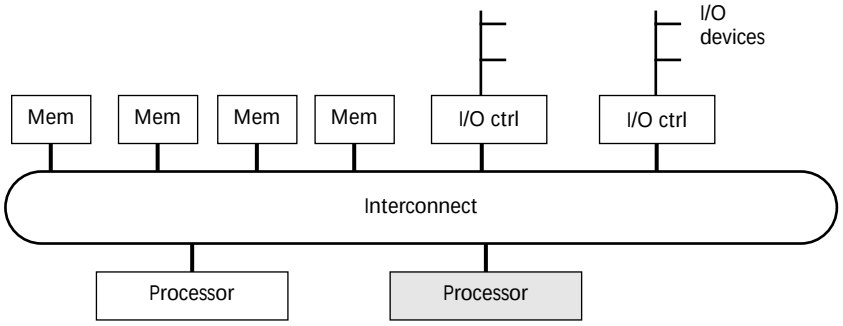
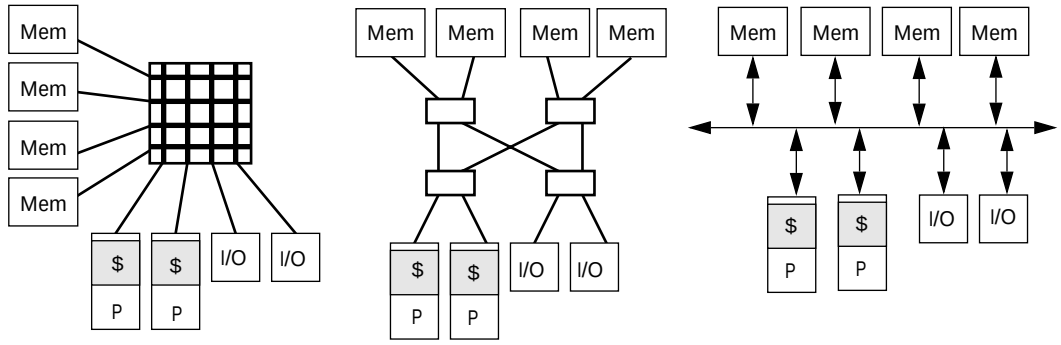


FIGURE 1.15 Extending a system into a shared memory multiprocessor by adding processor modules. Most systems consist of one or more memory modules accessible by a processor and I/O controllers through a hardware interconnect, typically a bus, crossbar, or multistage interconnect. Memory and I/O capacity are increased by attaching memory and I/O modules. Shared memory machines allow processing capacity to be increased by adding processor modules (shown as shaded).



(a) Crossbar switch (b) Multistage interconnection network (c) Bus interconnect

FIGURE 1.16 Typical shared memory multiprocessor interconnection schemes. The interconnection of multiple processors, with their local caches (indicated by $), and I/O controllers to multiple memory modules may be via crossbar, multistage interconnection network, or bus.

the number of ports. These savings come at the expense of increased latency and decreased bandwidth per port if all are used at once. The ability to access all memory directly from each processor has several advantages: any processor can run any process or handle any I/O event, and data structures can be shared within the operating system.

The widespread use of shared memory multiprocessor designs came about with the 32-bit microprocessor revolution in the mid-1980s because the processor, cache, floating-point, and memory management unit fit on a single board (Bell 1985) or even two to a board. Most mid-range machines, including minicomputers, servers, workstations, and personal computers, are organized around a central memory bus, as illustrated in Figure 1.16c, and the bus could be adapted to support multiple

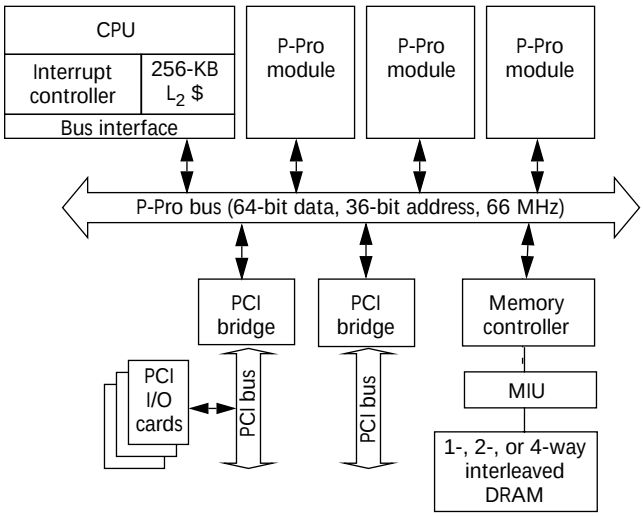


FIGURE 1.17(a) Physical and logical organization of the Intel Pentium Pro four-processor "quad pack." The Intel quad-processor Pentium Pro motherboard employed in many multiprocessor servers illustrates the major design elements of most small-scale shared memory multiprocessors. Its logical block diagram (a) shows that it can accommodate up to four processor modules, each containing a Pentium Pro processor, first-level caches, translation lookaside buffer, a 256-KB second-level cache, an interrupt controller, and a bus interface in a single chip connecting directly to a 64-bit memory bus. The bus operates at 66 MHz, and memory transactions are pipelined to give a peak bandwidth of 528 MB/s. A two-chip memory controller and four-chip memory interleave unit (MIU) connect the bus to multiple banks of DRAM. Bridges connect the memory bus to two independent PCI buses, which host display, network, SCSI, and lower-speed I/O connections. The Pentium Pro module contains all the logic necessary to support the multiprocessor communication architecture, including that required for memory and cache consistency. The structure of the Pentium Pro "quad pack" is similar to a large number of earlier SMP designs but has a much higher degree of integration and is targeted at a much larger volume. (b) shows an expanded view of a typical Pentium Pro SMP, an HP NetServer in the LX series. *Source:* Reproduced with permission of Hewlett-Packard Company.

processors. The standard bus access mechanism allows any processor to access any physical address in the system. Like the switch-based designs, all memory locations are equidistant to all processors, so all processors experience the same access time, or latency, on a memory reference. This configuration is usually called a *symmetric multiprocessor* (SMP).[4] SMPs are heavily used for execution of parallel programs as well as multiprogramming. The typical organization of the bus-based symmetric multiprocessor is illustrated in more detail by Figure 1.17, which describes the first

4. The term SMP is widely used but causes a bit of confusion. What exactly needs to be symmetric? Many designs are symmetric in some respect. The more precise description of what is intended by SMP is a shared memory multiprocessor where the cost of accessing a memory location is the same for all processors; that is, it has uniform access costs when the access actually is to memory. If the location is cached, the access will be faster, but cache access times and memory access times are the same on all processors.

FIGURE 1.17(b) Physical organization of the Intel Pentium Pro four-processor "quad pack"

highly integrated SMP for the commodity market. Figure 1.18 illustrates a high-end server organization that distributes the physical memory over the processor modules, but retains symmetric access.

The factors limiting the number of processors that can be supported with a bus-based organization are quite different from those in the switch-based approach. Adding processors to the switch is expensive; however, the aggregate bandwidth increases with the number of ports. The cost of adding a processor to the bus is small, but the aggregate bandwidth is fixed. Dividing this fixed bandwidth among the larger number of processors limits the practical scalability of the approach. (It is this critical bus bandwidth that is depicted in Figure 1.9.) Fortunately, caches reduce the bandwidth demand of each processor since many references are satisfied by the cache rather than by the memory. However, with data replicated in local caches, there is the potentially challenging problem of keeping the caches "consistent," which will be examined in detail in Chapters 5, 6, and 8.

Starting from a baseline of small-scale shared memory machines, illustrated in Figures 1.16–1.18, we may ask what is required to scale the design to a large number of processors. The basic processor component is well suited to the task since it is small and economical, but a problem clearly exists with the interconnect. The bus does not scale because it has a fixed aggregate bandwidth. The crossbar does not scale well because the cost increases as the square of the number of ports. Many alternative scalable interconnection networks exist, such that the aggregate bandwidth increases as more processors are added, but the cost does not become excessive. We need to be careful about the resulting increase in latency because the processor may stall while a memory operation moves from the processor to the memory module and back. If the latency of access becomes too large, the processors will spend much of their time waiting, and the advantages of more processors may be offset by poor utilization.

One natural approach to building scalable shared memory machines is to maintain the uniform memory access (or "dancehall") approach of Figure 1.15 and provide a scalable interconnect between the processors and the memories. Every memory access is translated into a message transaction over the network, much as it might be translated to a bus transaction in the SMP designs. The primary disadvantage of this approach is that the round-trip network latency is experienced on every memory access and a large bandwidth must be supplied to every processor.

An alternative approach is to interconnect complete processors, each with a local memory, as illustrated in Figure 1.19. In this nonuniform memory access (NUMA) approach, the local memory controller determines whether to perform a local memory access or a message transaction with a remote memory controller. Accessing local memory is faster than accessing remote memory. (The I/O system may either be a part of every node or consolidated into special I/O nodes, not shown.) Accesses to private data, such as code and stack, can often be performed locally, as can accesses to shared data that, by accident or intent, are stored on the local node. The ability to access the local memory quickly does not increase the time to access remote data

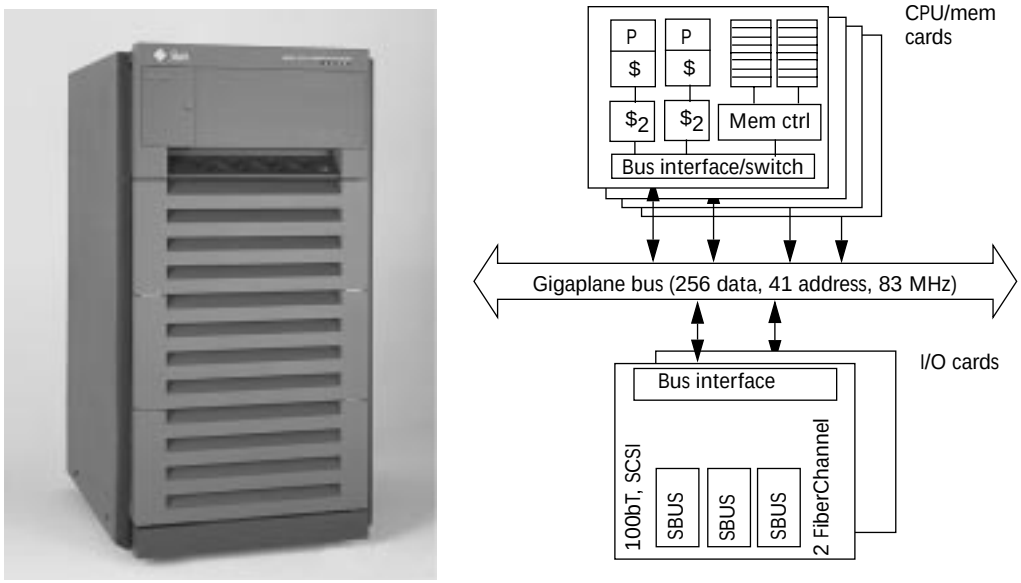


FIGURE 1.18 Physical and logical organization of the Sun Enterprise Server. A larger-scale design is illustrated by the Sun UltraSparc-based Enterprise multiprocessor server. The diagram shows its physical structure and logical organization. A wide (256-bit), highly pipelined memory bus delivers 2.5 GB/s of memory bandwidth. This design uses a hierarchical structure, where each card is either a complete dual processor with memory or a complete I/O system. The full configuration supports 16 cards of either type, with at least one of each. The CPU/mem card contains two UltraSparc processor modules, each with 16-KB level 1 and 512-KB level 2 caches, plus two 512-bit-wide memory banks and an internal switch. Thus, adding processors adds memory capacity and memory interleaving. The I/O card provides three SBUS slots for I/O extensions, a SCSI connector, a 100bT Ethernet port, and two FiberChannel interfaces. A typical complete configuration would be 24 processors and 6 I/O cards. Although memory banks are physically packaged with pairs of processors, all memory is equidistant from all processors and accessed over the common bus, preserving the SMP characteristics. Data may be placed anywhere in the machine with no performance impact. *Source:* The copyright for this photograph is owned by Sun Microsystems, Inc. and is used herein by permission.

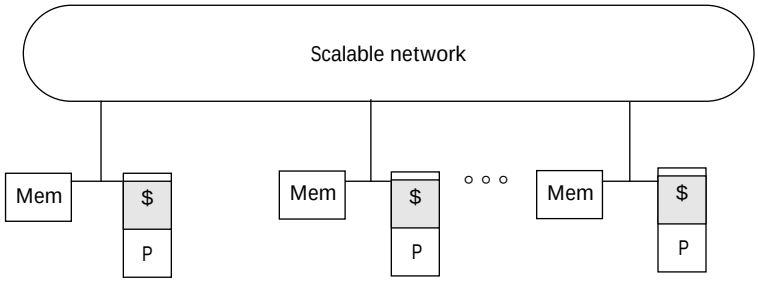


FIGURE 1.19 Nonuniform memory access (NUMA) scalable shared memory multiprocessor organization. Processor and memory modules are closely integrated such that access to local memory is faster than access to remote memories.

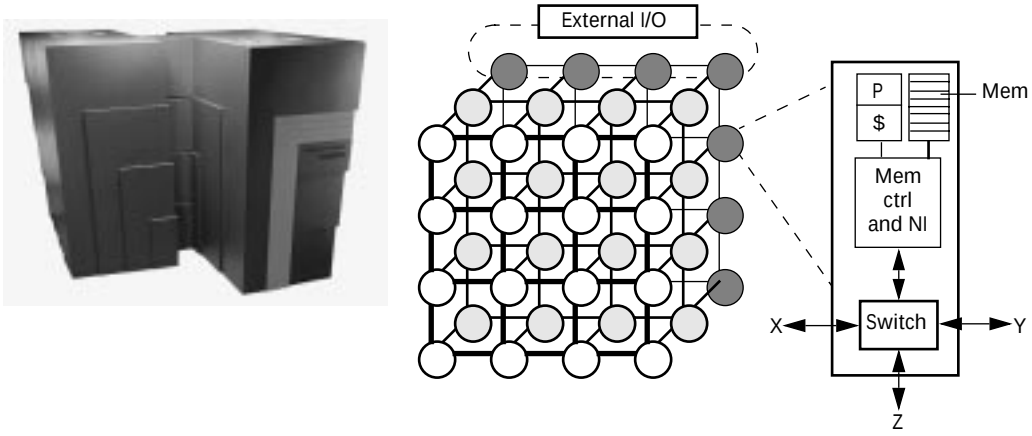


FIGURE 1.20 CRAY T3E scalable shared address space machine. The CRAY T3E is designed to scale up to a thousand processors supporting a global shared address space. Each node contains a DEC Alpha processor, local memory, a network interface integrated with the memory controller, and a network switch. The machine is organized as a three-dimensional cube, with each node connected to its six neighbors through 650-MB/s point-to-point links. Any processor can read or write any memory location; however, the NUMA characteristic of the machine is exposed in the communication architecture as well as in its performance characteristics. A short sequence of instructions is required to establish addressability to remote memory, which can then be accessed by conventional loads and stores. The memory controller captures the access to a remote memory and conducts a message transaction with the memory controller of the remote node on the local processor's behalf. The message transaction is automatically routed through intermediate nodes to the desired destination, with a small delay per "hop." The remote data is not cached since there is no hardware mechanism to keep it consistent. (We will look at other design points that allow shared data to be replicated throughout the processor caches.) The CRAY T3E I/O system is distributed over a collection of nodes on the surface of the cube, which are connected to the external world through an additional I/O network. *Source:* Photo courtesy of CRAY Research.

appreciably, so it reduces the average access time, especially when a large fraction of the accesses are to local data. The bandwidth demand placed on the network is also reduced. Although some conceptual simplicity arises from having all shared data equidistant from any processor, the NUMA approach has become far more prevalent at a large scale because of its inherent performance advantages and because it harnesses more of the mainstream processor memory system technology. One example of this style of design is the CRAY T3E, illustrated in Figure 1.20. This machine reflects the viewpoint that, although all memory is accessible to every processor, the distribution of memory across processors is exposed to the programmer. Caches are used only to hold data (and instructions) from local memory. It is the programmer's job to avoid frequent remote references. The SGI Origin is an example of a machine with a similar organizational structure, but it allows data from any memory to be replicated into any of the caches and provides hardware support to keep the caches consistent without relying on a bus connecting all the modules with a common set

of wires. While this book was being written, these two designs literally converged following the merger of the two companies.

To summarize, communication and cooperation in the shared address space programming model consists of reads and writes to shared variables; these operations are mapped directly to a communication abstraction consisting of load and store instructions accessing a global, shared address space, which is supported directly in hardware through access to shared physical memory locations. The programming model and communication abstraction are very close to the actual hardware. Each processor can *name* every physical location in the machine; a process can name all data it shares with others within its virtual address space. Data is transferred either as primitive types in the instruction set (bytes, words, etc.) or as cache blocks. Each process performs memory operations on addresses in its virtual address space; the address translation process identifies a physical location, which may be local or remote to the processor and may be shared with other processes. In either case, the hardware accesses it directly, without user or operating system software intervention. The address translation realizes protection within the shared address space, just as it does for uniprocessors, since a process can only access the data in its virtual address space.

The effectiveness of the shared memory approach depends on the latency incurred on memory accesses as well as the bandwidth of data transfer that can be supported. Just as a memory storage hierarchy allows data that is bound to an address to be migrated toward the processor, expressing communication in terms of the storage address space allows shared data to be migrated toward the processor that accesses it. However, migrating and replicating data across a general-purpose interconnect presents a unique set of challenges. We will see that to achieve scalability in such a design, the entire solution, including the hardware interconnect mechanisms used for maintaining the consistent shared memory abstractions, must scale well.

1.2.3 Message Passing

A second important class of parallel machines, called *message-passing architectures*, employs complete computers as building blocks—including the microprocessor, memory, and I/O system—and provides communication between processors as explicit I/O operations. The high-level block diagram for a message-passing machine is essentially the same as the NUMA shared memory approach shown in Figure 1.19. The primary difference is that communication is integrated at the I/O level rather than into the memory system. This style of design also has much in common with networks of workstations, or *clusters*, except that the packaging of the nodes is typically much tighter, there is no monitor or keyboard per node, and the network is of much higher capability than a standard local area network. The integration between the processor and the network tends to be much tighter than in traditional I/O structures, which support connection to devices that are much slower than the processor, since message passing is fundamentally processor-to-processor communication.

In message passing, a substantial distance exists between the programming model and the actual hardware primitives, with user communication performed through operating system or library calls that perform many lower-level actions, including the actual communication operation. Thus, our discussion of message passing begins with a look at the communication abstraction and then briefly surveys the evolution of hardware organizations supporting this abstraction.

The most common user-level communication operations on message-passing systems are variants of send and receive. In its simplest form, *send* specifies a local data buffer that is to be transmitted and a receiving process (typically on a remote processor). *Receive* specifies a sending process and a local data buffer into which the transmitted data is to be placed. Together, the matching send and receive cause a data transfer from one process to another, as indicated in Figure 1.21. In most message-passing systems, the send operation also allows an identifier or *tag* to be attached to the message, and the receiving operation specifies a matching rule (such as a specific tag from a specific processor, or any tag from any processor). Thus, the user program names local addresses and entries in an abstract process-tag space. *The combination of a send and a matching receive accomplishes a pairwise synchronization event and a memory-to-memory copy, where each end specifies its local data address.* There are several possible variants of this synchronization event, depending upon whether the send completes when the receive has been executed, when the send buffer is available for reuse, or when the request has been accepted. Similarly, the receive can potentially wait until a matching send occurs or simply post the receive. Each of these variants has somewhat different semantics and different implementation requirements.

Message passing has long been used as a means of communication and synchronization among arbitrary collections of cooperating sequential processes, even on a single processor. Important examples include programming languages, such as CSP and Occam, and common operating systems functions, such as sockets. Parallel programs using message passing are typically quite structured. Most often, all nodes execute identical copies of a program, with the same code and private variables. Usually, processes can name each other using a simple linear ordering of the processes comprising a program.

Early message-passing machines provided hardware primitives that were very close to the simple send/receive user-level communication abstraction, with some additional restrictions. A node was connected to a fixed set of neighbors in a regular pattern by point-to-point links that behaved as simple FIFOs (Seitz 1985). This sort of design is illustrated in Figure 1.22 for a small 3D cube. Many early machines were *hypercubes*, where each node is connected to n other nodes differing by one bit in the binary address, for a total of 2^n nodes. Others were *meshes*, where the nodes are connected to neighbors on two or three dimensions. The network topology was especially important in the early message-passing machines because only the neighboring processors could be named in a send or receive operation. The data transfer involved the sender writing to a link and the receiver reading from the link. The FIFOs were small and so the sender would not be able to finish writing the mes-

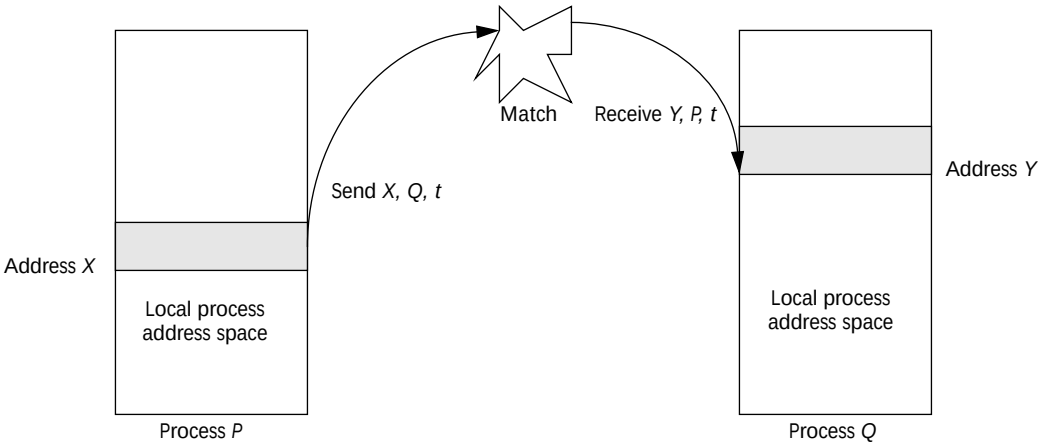


FIGURE 1.21 User-level send/receive message-passing abstraction. A data transfer from one local address space to another occurs when a send to a particular process is matched with a receive posted by that process.

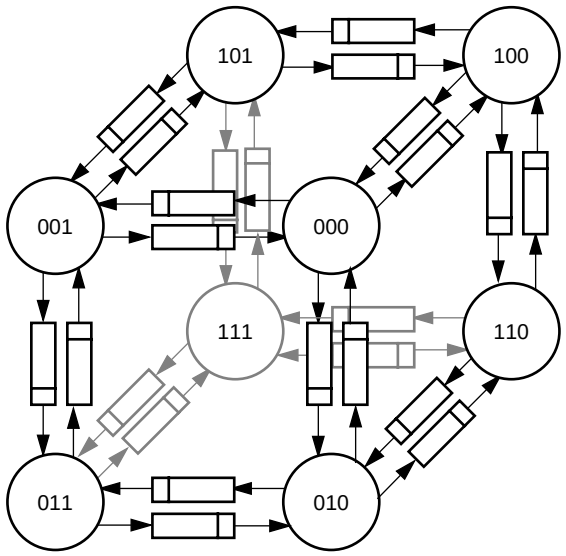


FIGURE 1.22 Typical structure of an early message-passing machine. Each node is connected to neighbors in three dimensions via FIFOs.

sage until the receiver started reading it, so the send would block until the receive occurred. (In modern terms, this is called *synchronous* message passing because the two events coincide in time.) The details of moving data were hidden from the

programmer in a message-passing library, forming a layer of software between send and receive calls and the actual hardware.[5]

The direct FIFO design was soon replaced by more versatile and more robust designs that provided *direct memory access* (DMA) transfers on either end of the communication event. A DMA device is a special-purpose controller that transfers data between memory and an I/O device without engaging the processor until the transfer is complete. The use of DMA allowed *nonblocking sends*, where the sender is able to initiate a send and continue with useful computation (or even perform a receive) while the send completes. On the receiving end, the transfer is accepted via a DMA transfer by the message layer into a buffer and queued until the target process performs a matching receive, at which point the data is copying into the address space of the receiving process.

The physical topology of the communication network so dominated the programming model of these early machines that parallel algorithms were often stated in terms of a specific interconnection topology, for example, a ring, a grid, or a hypercube (Fox et al. 1988). However, to make the machines more generally useful, the designers of the message layers provided support for communication between arbitrary processors rather than only between physical neighbors. This was originally supported by forwarding the data within the message layer along links in the network. Soon this routing function was moved into the hardware (as discussed in Chapter 10), so each node consisted of a processor with memory and a switch that could forward messages. However, in this approach, known as *store-and-forward*, the time to transfer a message is proportional to the number of hops it takes through the network, so an emphasis remained on interconnection topology. (See Exercise 1.7 for a brief store-and-forward example.)

The emphasis on network topology was significantly reduced with the introduction of more general-purpose networks, which pipelined the message transfer through each of the routers forming the interconnection network (Barton, Crownie, and McLaren 1994; Bomans and Roose 1989; Dunigan 1988; Homewood and McLaren 1993; Leiserson et al. 1996; Pierce and Regnier 1994; von Eicken et al. 1992). In most modern message-passing machines, the incremental delay introduced by each hop is small enough that the transfer time is dominated by the time to simply move that data between the processor and the network, not how far it travels (Groscup 1992; Homewood and McLaren 1993; Horiw et al. 1993; Pierce and Regnier 1994). This greatly simplifies the programming model; typically, the processors are viewed as simply forming a linear sequence with uniform communication costs. In other words, the communication abstraction reflects an organizational structure much as in Figure 1.19. One important example of such a machine is the IBM SP-2, illustrated in Figure 1.23, which is constructed from RS6000 workstation nodes, a scalable network, and a network interface containing a dedicated processor. Another

5. The motivation for synchronous message passing was not just from the machine structure; it was also present in important programming languages, especially CSP (Hoare 1978), because of its clean theoretical properties. Early in the microprocessor era, the approach was captured in a single-chip building block, the Transputer, which was widely touted during its development by INMOS as a revolution in computing.

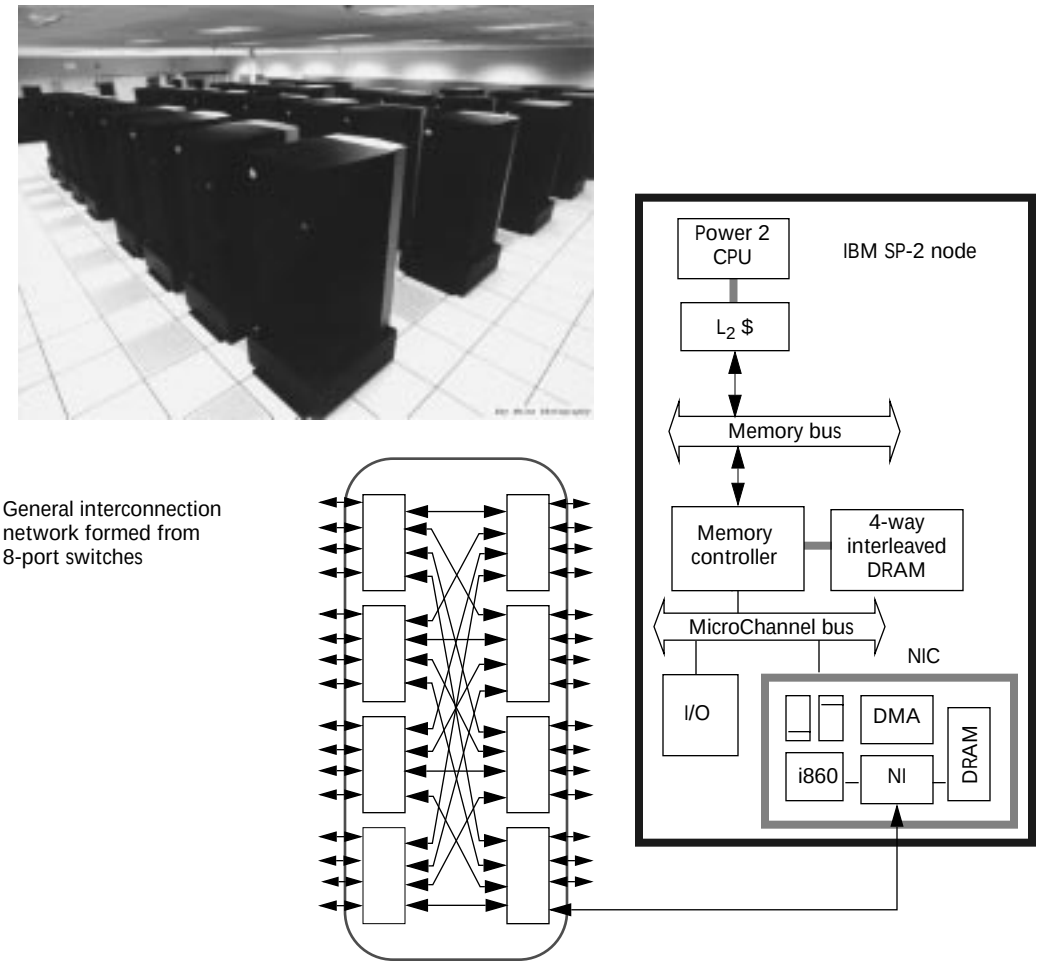


FIGURE 1.23 IBM SP-2 message-passing machine. The IBM SP-2 is a scalable parallel machine constructed essentially out of complete RS6000 workstations. Modest modifications are made to package the workstations into standing racks. A network interface card (NIC) is inserted at the MicroChannel I/O bus. The NIC contains the drivers for the actual link into the network, a substantial amount of memory to buffer message data, a direct memory access (DMA) engine, and a complete i860 microprocessor to move data between host memory and the network. The network itself is a butterfly-like structure, constructed by cascading 8×8 crossbar switches. The links operate at 40 MB/s in each direction, which is the full capability of the I/O bus. Several other machines employ a similar network interface design but connect directly to the memory bus rather than at the I/O bus. *Source:* Ray Mains Photography.

is the Intel Paragon, illustrated in Figure 1.24, which integrates the network interface more tightly to the processors in SMP nodes, where one of the processors is dedicated to supporting message passing.

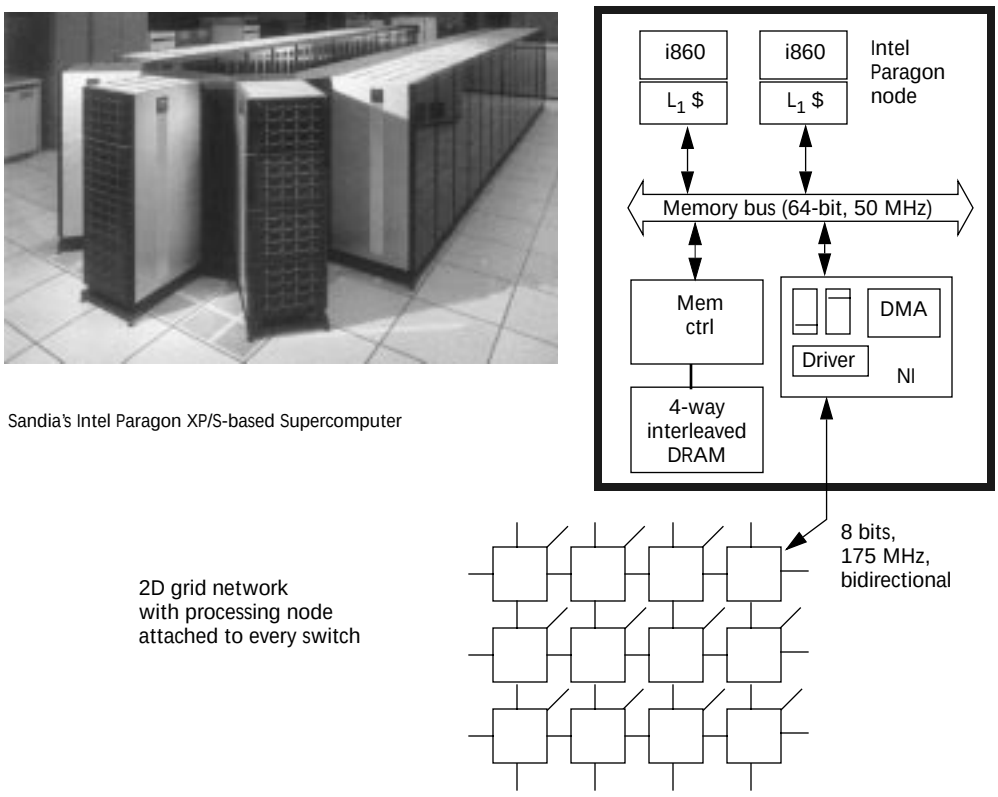


FIGURE 1.24 Intel Paragon. The Intel Paragon illustrates a much tighter packaging of nodes. Each card is an SMP with two or more i860 processors and a network interface chip connected to the cache-coherent memory bus. One of the processors is dedicated to servicing the network. In addition, the node has a DMA engine to transfer contiguous chunks of data to and from the network at a high rate. The network is a 3D grid, much like the CRAY T3E, with links operating at 175 MB/s in each direction. *Source:* Photo courtesy of Intel Corporation.

A processor in a message-passing machine can name only the locations in its local memory and each of the processors, perhaps by number or by route. A user process can only name private addresses and other processes; it can transfer data using the send/receive calls.

1.2.4 Convergence

Evolution of the hardware and software has blurred the once clear boundary between the shared memory and message-passing camps. First, consider the communication operations available to the user process.

- Traditional message-passing operations (send/receive) are supported on most shared memory machines through shared buffer storage. Send involves writing data, or a pointer to data, into the buffer; receive involves reading the data from shared storage. Flags or locks are used to control access to the buffer and to indicate events such as message arrival.
- On a message-passing machine, a user process may construct a global address space of sorts by carrying along pointers specifying the process and local virtual address in that process. Access to such a global address can be performed in software through an explicit message transaction. Most message-passing libraries allow a process to accept a message for any process, so each process can serve data requests from the others. A logical read is realized by sending a request to the process containing the object and receiving a response. The actual message transaction may be hidden from the user; it may be carried out by compiler-generated code for access to a shared variable.
- A shared virtual address space can be established on a message-passing machine at the page level. A collection of processes has a region of shared addresses but, for each process, only the pages that are local to it are accessible. Upon access to a missing (i.e., remote) page, a page fault occurs and the operating system engages the remote node in a message transaction to transfer the page and map it into the user address space.

At the level of machine organization, substantial convergence has occurred as well. Modern message-passing architectures appear essentially identical at the block diagram level to the scalable NUMA design illustrated in Figure 1.19. In the shared memory case, the network interface was integrated with the cache controller or memory controller in order for that device to observe cache misses and to conduct a message transaction to access memory in a remote node. In the message-passing approach, the network interface is essentially an I/O device. However, the trend has been to integrate this device more deeply into the memory system as well and to transfer data directly from and to the user address space. Some designs provide DMA transfers across the network, from memory on one machine to memory on the other machine, so the network interface is integrated fairly deeply with the memory system. Message passing is implemented on top of these remote memory copies (Barton, Crownie, and McLaren 1994). In some designs, a complete processor assists in communication, sharing a cache-coherent memory bus with the main processor (Groscup 1992; Pierce and Regnier 1994). Viewing the convergence from the other side, clearly all large-scale shared memory operations are ultimately implemented as message transactions at some level.

In addition to the convergence of scalable message-passing and shared memory machines, switch-based local area networks, including fast Ethernet, ATM, Fiber-Channel, and several proprietary designs (Boden et al. 1995; Gillett 1996) have emerged, providing scalable interconnects that are approaching what traditional parallel machines offer. These new networks are being used to connect collections of machines (which may be shared memory multiprocessors in their own right) into

clusters, which may operate as a parallel machine on individual large problems or as many individual machines on a multiprogramming load. Essentially all SMP vendors provide some form of network clustering to obtain better reliability.

In summary, message passing and a shared address space represent two clearly distinct programming models, each providing a well-defined paradigm for sharing, communication, and synchronization. However, the underlying machine structures have converged toward a common organization, represented by a collection of complete computers, augmented by a "communication assist" connecting each node to a scalable communication network. Thus, it is natural to consider supporting aspects of both in a common framework. Integrating the communication assist more tightly into the memory system tends to reduce the latency of network transactions and improve the bandwidth that can be supplied to or accepted from the network. We will want to look much more carefully at the precise nature of this integration and understand how it interacts with cache design, address translation, protection, and other traditional aspects of computer architecture.

1.2.5 Data Parallel Processing

A third important class of parallel machines has been variously called processor arrays, single-instruction-multiple-data machines, and data parallel architectures. The changing names reflect a gradual separation of the user-level abstraction from the machine operation. *The key characteristic of the data parallel programming model is that operations can be performed in parallel on each element of a large regular data structure, such as an array or matrix.* The program is logically a single thread of control, carrying out a sequence of either sequential or parallel steps. Within this general paradigm have been many novel designs, exploiting various technological opportunities, and considerable evolution as microprocessor technology has become such a dominant force.

An influential paper in the early 1970s (Flynn 1972) developed a taxonomy of computers, known as *Flynn's taxonomy*, which characterizes designs in terms of the number of distinct instructions issued at a time and the number of data elements they operate on: conventional sequential computers being *single-instruction-single-data* (SISD) and parallel machines built from multiple conventional processors being *multiple-instruction-multiple-data* (MIMD). The revolutionary alternative was *single-instruction-multiple-data* (SIMD). Its history is rooted in the mid-1960s when an individual processor was a cabinet full of equipment and an instruction fetch cost as much in time and hardware as performing the actual instruction. The idea was that all the instruction sequencing could be consolidated in the control processor. The data processors included only the ALU, memory, and a simple connection to nearest neighbors.

In the SIMD machines, the data parallel programming model was rendered directly in the physical hardware (Ball et al. 1962; Bouknight et al. 1972; Cornell 1972; Reddaway 1973; Slotnick, Borck, and McReynolds 1962; Slotnick 1967; Vick and Cornell 1978). Typically, a control processor broadcasts each instruction to an array of data processing elements (PEs), which are connected to form a regular grid,

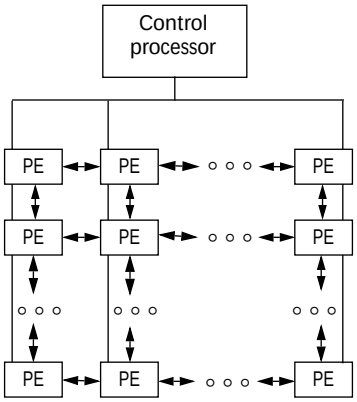


FIGURE 1.25 Typical organization of a data parallel (SIMD) machine. Individual processing elements (PEs) operate in lockstep under the direction of a single control processor. Traditionally, SIMD machines have provided a limited, regular interconnect among the PEs, although this was generalized in later machines, such as the Thinking Machines Corporation Connection Machine and the MasPar.

as suggested by Figure 1.25. It was observed that many important scientific computations involved uniform calculation on every element of an array or matrix, often involving neighboring elements in the row or column. Thus, the parallel problem data was distributed over the memories of the data processors, and scalar data was retained in the control processor's memory. The control processor instructed the data processors to each perform an operation on local data elements or to all perform a communication operation. For example, to average each element of a matrix with its four neighbors, a copy of the matrix would be shifted across the PEs in each of the four directions and a local accumulation performed in each PE. Data PEs typically included a condition flag, allowing some to abstain from an operation. In some designs, the local address could be specified with an indirect addressing mode, allowing all processors to do the same operation but with different local data addresses.

The development of arrays of processors was almost completely eclipsed in the mid-1970s with the development of vector processors. In these machines, a scalar processor is integrated with a collection of function units that operate on vectors of data out of one memory in a pipelined fashion. The ability to operate on vectors anywhere in memory eliminated the need to map application data structures onto a rigid interconnection structure and greatly simplified the problem of getting data aligned so that local operations could be performed. The first vector processor, the CDC Star-100, provided vector operations in its instruction set that combined two source vectors from memory and produced a result vector in memory. The machine only operated at full speed if the vectors were contiguous, and hence a large fraction of the execution time was spent simply transposing matrices. A dramatic change occurred in 1976 with the introduction of the CRAY-1, which extended the concept of a load-store architecture employed in the CDC 6600 and CDC 7600 (and rediscovered in modern RISC machines) to apply to vectors. Vectors in memory, of any fixed stride, were transferred to or from contiguous vector registers by vector load and store instructions. Arithmetic was performed on the vector registers. The use of a very fast scalar processor (operating at the unprecedented rate of 80 MHz), tightly

integrated with the vector operations and utilizing a large semiconductor memory rather than core, took over the world of supercomputing. Over the next twenty years, CRAY Research led the supercomputing market by increasing the bandwidth for vector memory transfers, the number of processors, the number of vector pipelines, and the length of the vector registers, resulting in the performance growth indicated in Figures 1.10 and 1.11.

The SIMD data parallel machine experienced a renaissance in the mid-1980s, as VLSI advances made simple 32-bit processors just barely practical (Batcher 1974, 1980; Hillis 1985; Nickolls 1990; Tucker and Robertson 1988). The unique twist in the data parallel regime was to place 32 very simple 1-bit processing elements on each chip, along with serial connections to neighboring processors, while consolidating the instruction sequencing capability in the control processor. In this way, systems with several thousand bit-serial processing elements could be constructed at reasonable cost. In addition, it was recognized that the utility of such a system could be increased dramatically with the provision of a general interconnect allowing an arbitrary communication pattern to take place in a single, rather long step, in addition to the regular grid neighbor connections (Hillis 1985; Hillis and Steele 1986; Nickolls 1990). The sequencing mechanism that expanded conventional integer and floating-point operations into a sequence of bit-serial operations also provided a means of "virtualizing" the processing elements, so that a few thousand processing elements could give the illusion of operating in parallel on millions of data elements with one virtual PE per data element.

The technological factors that made this bit-serial design attractive also provided fast, inexpensive, single-chip floating-point units and rapidly gave way to very fast microprocessors with integrated floating point and caches. This eliminated the cost advantage of consolidating the sequencing logic and provided equal peak performance on a much smaller number of complete processors. The simple, regular calculations on large matrices that motivated the data parallel approach also have tremendous spatial and temporal locality (if the computation is properly mapped onto a smaller number of complete processors), with each processor responsible for a large number of logically contiguous data points. Caches and local memory can be brought to bear on the set of data points local to each node while communication occurs across the boundaries or as a global rearrangement of data.

Thus, while the user-level abstraction of parallel operations on large regular data structures continued to offer an attractive solution to an important class of problems, the machine organization employed with data parallel programming models evolved toward a more generic parallel architecture of multiple cooperating microprocessors, much like scalable shared memory and message-passing machines, although several designs maintain specialized network support for global synchronization. One such example of network support is for a *barrier*, which causes each process to wait at a particular point in the program until all other processes have reached that point (Horiw et al. 1993; Leiserson et al. 1996; Kumar 1992; Kessler and Schwarzmeier 1993; Koeninger, Furtney, and Walker 1994). Indeed, the SIMD approach evolved into the SPMD (single-program-multiple-data) approach, in

which all processors execute copies of the same program, and has thus largely converged with the more structured forms of shared memory and message-passing programming.

Data parallel programming languages are usually implemented by viewing the local address spaces of a collection of processes, one per processor, as forming an explicit global address space. Data structures are laid out across this global address space with a simple mapping from indexes to processor and local offset. The computation is organized as a sequence of "bulk synchronous" phases of either local computation or global communication, separated by a global barrier (Valiant 1990). Because all processors perform communication together and share a global view of what is going on, either a shared address space or message passing can be employed. For example, if a phase involved every processor doing a write to an address in the processor "to the left," it could be realized by each doing a send to the left and a receive "from the right" into the destination address. Similarly, every processor doing a read can be realized by every processor sending the address and then every processor sending back the data. In fact, the code that is produced by compilers for modern data parallel languages is essentially the same as for the structured control-parallel programs that are most common in shared memory and message-passing programming models. The convergence in machine structure has been accompanied by a convergence in how the machines are actually used.

1.2.6 Other Parallel Architectures

The mid-1980s renaissance gave rise to several other architectural directions that received considerable investigation by academia and industry, but enjoyed less commercial success than the three classes just discussed and therefore experienced less use as a vehicle for parallel programming. Two approaches that were developed into complete programming systems were dataflow architectures and systolic architectures. Both represent important conceptual developments of continuing value as the field evolves.

Dataflow Architecture

Dataflow models of computation sought to make the essential aspects of a parallel computation explicit at the machine level, without imposing artificial constraints that would limit the available parallelism in the program. The idea is that the program is represented by a graph of essential data dependences, as illustrated in Figure 1.26, rather than as a fixed collection of explicitly sequenced threads of control. An instruction may execute whenever its data operands are available. The graph may be spread arbitrarily over a collection of processors. Each node specifies an operation to perform and the address of each of the nodes that need the result. In the original form, a processor in a dataflow machine operates as a simple circular pipeline. A message, or *token*, from the network consists of data and an address, or *tag*, of its destination node. The tag is compared against those in a matching store. If

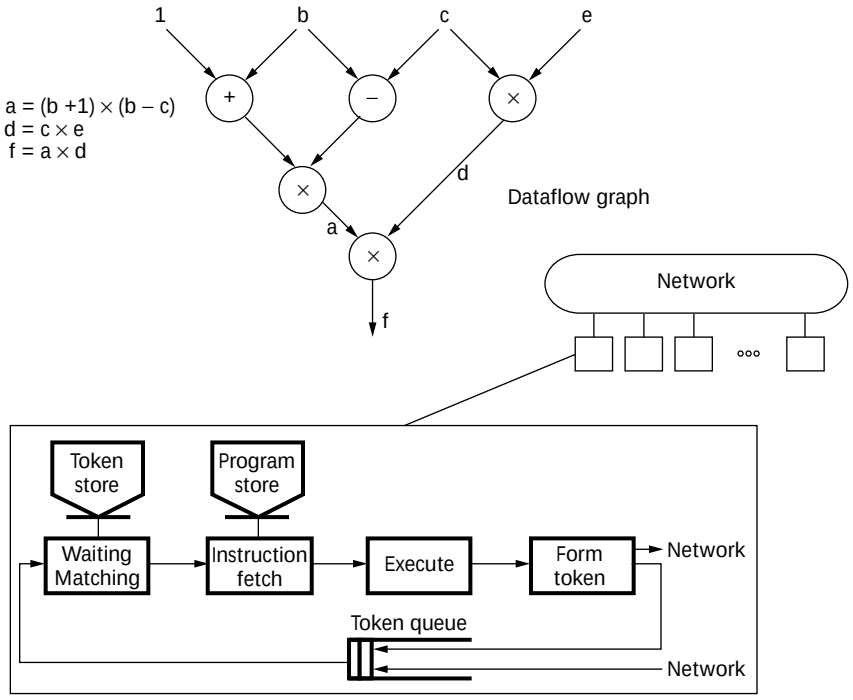


FIGURE 1.26 Dataflow graph and basic execution pipeline. A node in the graph fires when operands are present on its input. It produces results on its outputs that are delivered to adjacent nodes in the graph. The execution pipeline implements this firing rule by detecting when matching data tokens are present, fetching the corresponding instruction, performing the operation, and forming result tokens.

present, the matching token is extracted and the instruction is issued for execution. If not, the token is placed in the store to await its partner. When a result is computed, a new message or token containing the result data is sent to each of the destinations specified in the instruction. The same mechanism can be used whether the successor instructions are local or on a remote processor.

The primary division within dataflow architectures is whether the graph is *static*, with each node representing a primitive operation, or *dynamic*, in which case a node can represent the invocation of an arbitrary function, itself represented by a graph. In dynamic, or *tagged-token*, architectures, the effect of dynamically expanding the graph on function invocation is usually achieved by carrying additional context information in the tag, rather than actually modifying the program graph.

The key characteristics of dataflow architectures are the ability to name operations performed anywhere in the machine, the support for synchronization of independent operations, and dynamic scheduling at the machine level. As the dataflow machine designs matured into real systems programmed in high-level parallel

languages, a more conventional structure emerged. Typically, parallelism was generated in the program as a result of parallel function calls and parallel loops, so it was attractive to allocate these larger chunks of work to processors. This led to a family of designs organized essentially like the NUMA design of Figure 1.19, the key differentiating features being direct support for a large, dynamic set of threads of control and the integration of communication with thread generation. The network is closely integrated with the processor; in many designs, the "current message" is available in special registers, and hardware support is available for dispatching to a thread identified in the message. In addition, many designs provide extra state bits on memory locations in order to provide fine-grained synchronization (i.e., synchronization on an element-by-element basis) rather than using locks to synchronize accesses to an entire data structure. In particular, each message could schedule a chunk of computation that could make use of local registers and memory.

By contrast, in shared memory machines, the generally adopted view is that a static or slowly varying set of processes operates within a shared address space, so the compiler or program maps the logical parallelism in the program to a set of processes by assigning loop iterations, maintaining a shared work queue, or the like. Similarly, message-passing programs involve a static, or nearly static, collection of processes that can name one another in order to communicate. In data parallel architectures, the compiler or sequencer maps a large set of "virtual processor" operations onto processors by assigning iterations of a regular loop nest. In the dataflow case, the machine provides the ability to name a very large and dynamic set of threads that can be mapped arbitrarily to processors. Typically, these machines provide a global address space as well. As was the case with message-passing and data parallel machines, dataflow architectures experienced a gradual separation of programming model and hardware structure as the approach matured.

Systolic Architectures

Another novel approach was *systolic architectures*, which sought to replace a single sequential processor by a regular array of simple processing elements and, by carefully orchestrating the flow of data between PEs, obtain very high throughput with modest memory bandwidth requirements. These designs differ from conventional pipelined function units in that the array structure can be nonlinear (e.g., hexagonal), the pathways between PEs may be multidirectional, and each PE may have a small amount of local instruction and data memory. They differ from SIMD in that each PE might do a different operation.

The early proposals were driven by the opportunity offered by VLSI to provide inexpensive special-purpose chips. A given algorithm could be represented directly as a collection of specialized computational units connected in a regular, space-efficient pattern. Data would move through the system at regular "heartbeats" as determined by local state. Figure 1.27 illustrates a design for computing convolutions using a simple linear array. At each beat the input data advances to the right, is multiplied by a local weight, and is accumulated into the output sequence as it also

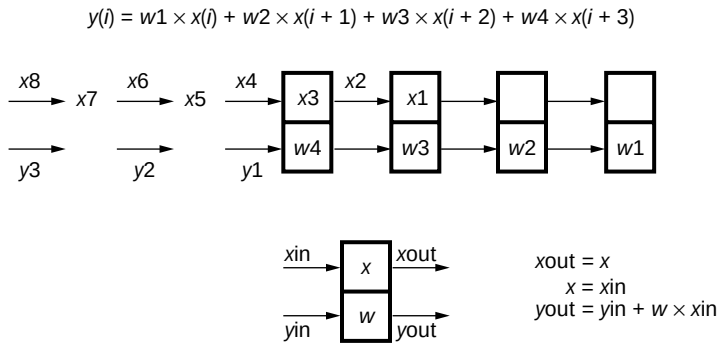


FIGURE 1.27 Systolic array computation of an inner product. Each box represents a computational unit performing a specific function. Every time the clock beats, all units accept inputs, compute results, and generate outputs. Data moves through the systolic array with each beat.

advances to the right. The systolic approach has aspects in common with message-passing, data parallel, and dataflow models but takes on a unique character for a specialized class of problems.

Practical realizations of these ideas, such as iWarp (Borkar et al. 1990), provided quite general programmability in the nodes, so that a variety of algorithms could be realized on the same hardware. The key differentiation is that the network can be configured as a collection of dedicated channels, representing the systolic communication pattern, and data can be transferred directly from processor registers to processor registers across a channel. The global knowledge of the communication pattern is exploited to reduce contention and even to avoid deadlock. The key characteristic of systolic architectures is the ability to integrate highly specialized computation under simple, regular, and highly localized communication patterns.

Systolic algorithms have also been generally amenable to solutions on generic machines, using the fast barrier to delineate coarser-grained phases. The regular, local communication pattern of these algorithms yields good locality when large portions of the logical systolic array are executed on each process, the communication bandwidth needed is low, and the synchronization requirements are simple. Thus, these algorithms have proved effective on the entire spectrum of parallel machines.

1.2.7 A Generic Parallel Architecture

In examining the evolution of the major approaches to parallel architecture, we see a clear convergence for scalable machines toward a generic parallel machine organization, illustrated in Figure 1.28. The machine comprises a collection of essentially complete computers, each with one or more processors and memory, connected through a scalable communication network via *communication assist*—a controller

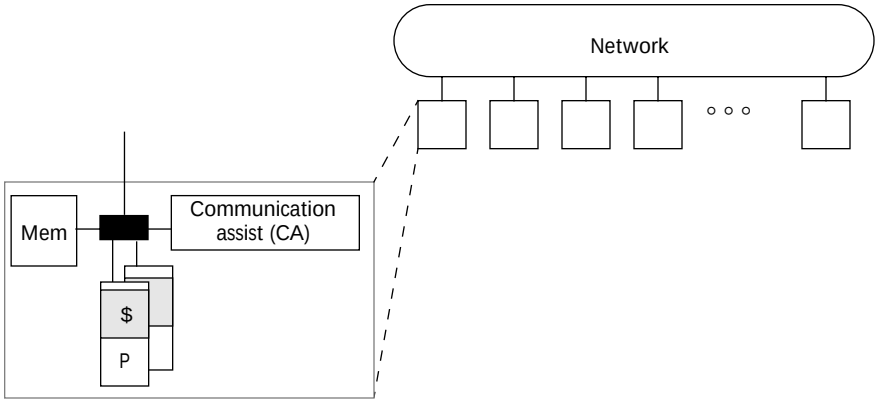


FIGURE 1.28 Generic scalable multiprocessor organization. A collection of essentially complete computers, including one or more processors and memory, communicating through a general-purpose, high-performance, scalable interconnect. Typically, each node contains a controller that assists in communication operations across the network.

or auxiliary processing unit that assists in generating outgoing messages or handling incoming messages. While the consolidation within the field may seem to narrow the design space, in fact, great diversity and debate remains, centered on what functionality should be provided within the assist and how it interfaces to the processor, memory system, and network. Recognizing that these are specific differences within a largely similar organization helps us to understand and evaluate the important organizational trade-offs.

Not surprisingly, different programming models place different requirements on the design of the communication assist and influence which operations are common and should be optimized. In the shared memory case, the assist is tightly integrated with the memory system in order to capture the memory events that may require interaction with other nodes. The assist must also accept messages and perform memory operations and state transitions on behalf of other nodes. In the message-passing case, communication is initiated by explicit actions, either at the system or user level, so it is not required that memory system events be observed. Instead, a need exists to initiate the messages quickly and to respond to incoming messages. The response may require that a tag match be performed, that buffers be allocated, that data transfer commence, or that an event be posted. The data parallel and systolic approaches place an emphasis on fast global synchronization, which may be supported directly in the network or in the assist. Dataflow places an emphasis on fast dynamic scheduling of computation based on an incoming message. Systolic algorithms present the opportunity to exploit global patterns in local scheduling. Even with these differences, it is important to observe that all of these approaches share common aspects; they need to initiate network transactions as a result of specific processor events, and they need to perform simple operations on the remote node to carry out the desired event.

We also see that a separation has emerged between programming model and machine organization as parallel programming environments have matured. For example, Fortran 90 and High Performance Fortran provide a shared address, data parallel programming model that is implemented on a wide range of machines—some supporting a shared physical address space, others with only message passing. The compilation techniques for these machines differ radically, even though the machines appear organizationally similar, because of differences in communication and synchronization operations provided in the communication abstraction and vast differences in the performance characteristics of these operations. As a second example, popular message-passing libraries, such as PVM (parallel virtual machine) and MPI (message-passing interface), are implemented on this same range of machines, but the implementation of the libraries differs dramatically from one kind of machine to another. The same observations hold for parallel operating systems.

1.3 FUNDAMENTAL DESIGN ISSUES

Given how the state of the art in parallel architecture has advanced, we need to take a fresh look at how to organize the body of material in the field. Traditional machine taxonomies, such as SIMD/MIMD, are of little help since multiple general-purpose processors are so dominant. We cannot focus entirely on programming models since in many cases widely differing machine organizations support a common programming model. We cannot just look at hardware structures either, since common elements are employed in many different ways. Instead, we should focus our attention on the architectural distinctions that make a difference to the software that is to run on the machine. In particular, we need to highlight those aspects that influence how a compiler would generate code from a high-level parallel language, how a library writer would code a well-optimized library, or how an application would be written in a low-level parallel language. We can then approach the design problem as one that is constrained from above by how programs use the machine and from below by what the basic technology can provide.

The guiding principles presented in this book for understanding modern parallel architecture are indicated by the layers of abstraction shown in Figure 1.13. Fundamentally, we must understand the operations that are provided at the user-level communication abstraction, how various programming models are mapped to these primitives, and how these primitives are mapped to the actual hardware. Excessive emphasis on the high-level programming model without attention to how it can be mapped to the machine would detract from understanding the fundamental architectural issues, as would excessive emphasis on the specific hardware mechanisms in each particular machine.

This section looks more closely at the communication abstraction and the basic requirements of a programming model. It then defines more formally the key concepts that tie the layers together: naming, ordering, and communication and replication of data. Finally, it introduces the basic performance models required to resolve design trade-offs.

1.3.1 Communication Abstraction

The communication abstraction forms the key interface between the programming model and the system implementation. It plays a role very much like the instruction set in conventional sequential computer architecture. Viewed from the software side, it must have a precise, well-defined meaning so that the same program will run correctly on many implementations. In addition, the operations provided at this layer must be simple, composable entities with clear costs, so that the software can be optimized for performance. Viewed from the hardware side, it must also have a well-defined meaning so that the machine designer can determine where performance optimizations can be performed without violating the software assumptions. While the abstraction needs to be precise, the machine designer would like it not to be overly specific, so it does not prohibit useful techniques for performance enhancement or frustrate efforts to exploit properties of newer technologies.

The communication abstraction is, in effect, a contract between the hardware and the software allowing each the flexibility to improve what it does while working correctly together. To understand the "terms" of this contract, we need to look more carefully at the basic requirements of a programming model.

1.3.2 Programming Model Requirements

A parallel program consists of one or more threads of control operating on data. A *parallel programming model* specifies what data can be *named* by the threads, what *operations* can be performed on the named data, and what *ordering* exists among these operations.

To make these issues concrete, consider the programming model for a uniprocessor. A thread can name the locations in its virtual address space and can name machine registers. In some systems, the address space is broken up into distinct code, stack, and heap segments whereas in others it is flat. Similarly, different programming languages provide access to the address space in different ways; for example, some allow pointers and dynamic storage allocation, others do not. Regardless of these variations, the instruction set provides the operations that can be performed on the named locations. For example, in RISC machines the thread can load data from or store data to memory but can perform arithmetic and comparisons only on data in registers. Older instruction sets support arithmetic on either. Compilers typically mask these differences at the hardware/software boundary, so the user's programming model is one of performing operations on variables that hold data. The hardware translates each virtual address to a physical address on every operation.

The ordering among memory operations is *sequential program order*. The programmer's view is that variables are read and modified in the top-to-bottom, left-to-right order specified in the program. More precisely, the value returned by a read to an address is the last value written to the address in the sequential execution order of the program. This ordering assumption is essential to the logic of the program. However, the reads and writes may not actually be performed in program order

because the compiler performs optimizations when translating the program to the instruction set and the hardware performs optimizations when executing the instructions. Both make sure the program cannot tell that the order has been changed. The compiler and hardware preserve the *dependence order*, that is, if a variable is written and then read later in the program order, they make sure that the later operation uses the proper value, but they may avoid actually writing and reading the value to and from memory or may defer the write until later. Collections of reads with no intervening writes may be completely reordered and, generally, writes to different addresses can be reordered as long as dependences from intervening reads are preserved. This reordering occurs at the compilation level, for example, when the compiler allocates variables to registers, manipulates expressions to improve pipelining, or transforms loops to reduce overhead and improve the data access pattern. It occurs at the machine level when instruction execution is pipelined, multiple instructions are issued per cycle, or write buffers are used to hide memory latency. We depend on these optimizations for performance. They work because for the program to observe the effect of a write, it must read the variable; this creates a dependence, which is preserved. Thus, the illusion of program order is preserved while actually executing the program in the weaker dependence order.[6] We operate in a world where essentially all programming languages embody a programming model of sequential order of operations on variables in a virtual address space, and the system enforces a weaker order wherever it can do so without changing the results of the program.

Now let's return to parallel programming models. The informal discussion earlier in this chapter indicated the distinct positions adopted on naming, operation set, and ordering. Naming and operation set are what typically characterize the models; however, ordering is of key importance. A parallel program must coordinate the activity of its threads to ensure that the dependences within the program are enforced; *this requires explicit synchronization operations when the ordering implicit in the basic operations is not sufficient*. As architects (and compiler writers), we need to understand the ordering properties to see what optimization "tricks" we can play for performance. We can focus on shared address and message-passing programming models since they are the most widely used; other models, such as data parallel, are usually implemented in terms of one of them.

The shared address space programming model assumes one or more threads of control, each operating in an address space that contains a region shared between threads, and may contain a region that is private to each thread. Typically, the shared region is shared by all threads. All the operations defined on private addresses are defined on shared addresses; in particular, the program accesses and updates shared variables simply by using them in expressions and assignment statements.

Message-passing models assume a collection of processes each operating in a private address space and each able to name the other processes. The normal unipro-

---

6. The illusion breaks down a little bit for system programmers, say, if the variable is actually a control register on a device. Then the actual program order must be preserved. This is usually accomplished by flagging the variable as special; for example, using the volatile type modifier in C.

cessor operations are provided on the private address space, in program order. The additional operations, send and receive, operate on the local address space and the global process space. Send transfers data from the local address space to a process. Receive accepts data into the local address space from a process. Each send/receive pair is a specific point-to-point synchronization operation. Many message-passing languages offer global, or *collective*, communication operations as well, such as broadcast.

Naming

The position adopted on naming in the programming model is presented to the programmer through the programming language or programming environment. It is what the logic of the program is based upon. However, the issue of naming is critical at each level of the communication architecture. Certainly one possible strategy is to have the operations in the programming model be one to one with the operations in the communication abstraction at the user/system boundary and to have these be one to one with the hardware primitives. However, it is also possible for the compiler and libraries to provide a level of translation between the programming model and the communication abstraction, or for the operating system to intervene to handle some of the operations at the user/system boundary. These alternatives allow the architect to consider implementing the common, simple operations directly in hardware and supporting the more complex operations partly or wholly in software.

Let us consider the ramifications of naming at the layers using the two primary programming models: shared address and message passing. First, in a shared address model, accesses to shared variables in the program are usually mapped by the compiler to load and store instructions on shared virtual addresses, just like access to any other variable. This is not the only option, however. The compiler could generate special code sequences for accesses to shared variables. A machine supports a *global physical address space* if any processor is able to generate a physical address for any location in the machine and access the location in a single memory operation. It is straightforward to realize a shared virtual address space on a machine providing a global physical address space: establish the virtual-to-physical mapping so that shared virtual addresses map to the same physical location (i.e., the processes have the same entries in their page tables). However, the existence of the level of translation allows for other approaches. A machine supports *independent local physical address spaces* if each processor can only access a distinct set of locations. Even on such a machine, a shared virtual address space can be provided by mapping virtual addresses that are local to a process to the corresponding physical address. The nonlocal addresses are left unmapped so upon access to a nonlocal shared address a page fault will occur, allowing the operating system to intervene and access the remote shared data. Although this approach can provide the same naming, operations, and ordering to the program, it clearly has different hardware requirements at the hardware/software boundary. The architect's job is to resolve these design trade-offs across layers of the system implementation so that the result is efficient and cost-effective for the target application workload on available technology.

Second, message-passing operations could be realized directly in hardware, but the matching and buffering aspects of the send/receive operations are better suited to software implementation. More basic data transport primitives are well supported in hardware. Thus, in essentially all parallel machines, the message-passing programming model is realized via a software layer that is built upon a simpler communication abstraction. At the user/system boundary, one approach is to have all message operations go through the operating system as if they were I/O operations. However, the frequency of message operations is much greater than I/O operations, so it makes sense to use the operating system support to set up resources, privileges, and so on and allow the frequent, simple data transfer operations to be supported directly in hardware. On the other hand, we might consider adopting a shared virtual address space as the lower-level communication abstraction, in which case send and receive operations involve writing and reading shared buffers and posting the appropriate synchronization events.

The issue of naming arises at each level of abstraction in a parallel architecture, not just in the programming model. As architects, we need to design against the frequency and type of operations that occur at the communication abstraction, understanding that the trade-offs at this boundary involve what is supported directly in hardware and in software.

Operations

Each programming model defines a specific set of operations that can be performed on the data or objects that can be named within the model. For the case of a shared address model, these include reading and writing shared variables as well as various atomic read-modify-write operations on shared variables, which are used to synchronize the threads. For message passing, the operations are send and receive on private (local) addresses and process identifiers, as described previously. Each element of data in the program is named by a process number and a local address within the process. A message-passing model does define a global address space of sorts. However, no operations are defined on these global addresses. They can be passed around and interpreted by the program, for example, to emulate a shared address style of programming on top of message passing, but they cannot be operated on directly at the communication abstraction. As architects, we need to be aware of the operations defined at each level of abstraction. In particular, we need to be very clear on what ordering among operations is assumed to be present at each level of abstraction, where communication takes place, and how data is replicated.

Ordering

The properties of the specified order among operations have a profound effect throughout the layers of parallel architecture. Notice, for example, that the message-passing model places no assumption on the ordering of operations by distinct processes except the explicit program order associated with the send/receive operations,

whereas a shared address model must specify aspects of how processes see the order of operations performed by other processes. Ordering issues are important and rather subtle. Many of the tricks that we play for performance in the uniprocessor context involve relaxing the order assumed by the programmer to gain performance, either through parallelism or improved locality or both. Exploiting parallelism and locality is even more important in the multiprocessor case. Thus, we need to understand what new tricks can be played. We also need to examine which of the old tricks are still valid. Can we perform the traditional sequential optimizations at the compiler and architecture level on each process of a parallel program? Where can the explicit synchronization operations be used to allow ordering to be relaxed on the conventional operations? To answer these questions, we need to develop a much more complete understanding of how programs use the communication abstraction, what properties they rely upon, and what machine structures we would like to exploit for performance.

A natural position to adopt on ordering is that operations in a thread are in program order. That is what the programmer would assume for the special case of one thread. However, there remains the question of what ordering can be assumed among operations performed on shared variables by different threads. The threads operate independently and, potentially, at different speeds so no clear notion of "latest" is defined. If we have in mind that the machines behave as a collection of simple processors operating on a common, centralized memory, then it is reasonable to expect the global order of memory accesses to be some arbitrary interleaving of the individual program orders. In reality we won't build the machines this way, but it establishes what operations are implicitly ordered by the basic operations in the model. This interleaving is also what we expect of a collection of threads that are time-shared, perhaps at a very fine level, on a uniprocessor.

Where the implicit ordering is not enough, explicit synchronization operations are required. Parallel programs require two types of synchronization:

- *Mutual exclusion* ensures that certain operations on certain data are performed by only one thread or process at a time. We can imagine a room that must be entered to perform such an operation, and only one process can be in the room at a time. This is accomplished by locking the door upon entry and unlocking it on exit. If several processes arrive at the door together, only one will get in and the others will wait until it leaves. The order in which the processes are allowed to enter does not matter and may vary from one execution of the program to the next; what matters is that they do so one at a time. Mutual exclusion operations tend to serialize the execution of processes.
- *Events* are used to inform other processes that some point of execution has been reached so that they can proceed knowing that certain dependences have been satisfied. These operations are like passing a baton from one runner to the next in a relay race or the starter firing a gun to indicate the start of a race. If one process writes a value that another is supposed to read, an event synchronization operation must take place to indicate that the value is ready to be read. Events may be *point-to-point*, involving a pair of processes, or they may be *global*, involving all processes or a group of processes.

1.3.3 Communication and Replication

The final issues that are closely tied to the layers of parallel architecture are communication and data replication. Communication and replication are inherently related. Consider first a message-passing operation. The effect of the send/receive pair is to copy data that is in the sender's address space into a region of the receiver's address space. This transfer is essential for the receiver to access the data. If the data is produced by the sender, it reflects a *true communication* of information from one process to the other. If the data just happens to be stored at the sender, perhaps because that was the initial configuration of the data or because the data set was simply too large to fit on any one node, then this transfer merely makes a replica of the data where it is used. In this case, the processes are not actually communicating information from one to another via the data transfer. If the data were replicated or positioned properly over the processes to begin with, there would be no need to communicate it in a message. More importantly, if the receiver uses the data over and over again, it can reuse its replica without additional data transfers. The sender can modify the region of addresses that was previously communicated with no effect on the previous receiver. If the effect of these later updates is to be communicated, an additional transfer must occur.

Consider now a conventional data access on a uniprocessor through a cache. If the cache does not contain the desired address, a miss occurs and the block is transferred from the memory that serves as a backing store. The data is implicitly replicated into the cache near the processor that accesses it. If the processor reuses the data while it resides in the cache, further transfers with the memory are avoided. In the uniprocessor case, the processor produces the data and the processor consumes it, so the "communication" with the memory occurs only because the data does not fit in the cache or is being accessed for the first time.

Interprocess communication and data transfer within the storage hierarchy become melded together in a shared physical address space. Cache misses cause a data transfer across the machine interconnect whenever the physical backing storage for an address is remote to the node accessing the address, whether the address is private or shared and whether the transfer is a result of true communication or just a data access. The natural tendency of the machine is to replicate data into the caches of the processors that access the data. If the data is reused while it is in the cache, no data transfers occur; this is a major advantage. However, when a write to shared data occurs, something must be done to ensure that later reads by other processors get the new data rather than the old data that was replicated into their caches. This will involve more than a simple data transfer.

To be clear on the relationship of communication and replication, it is important to distinguish several concepts that are frequently bundled together. When a program performs a write, it binds a data value to an address; a read obtains the data value bound to an address. The data resides in some physical storage element in the machine. A *data transfer* occurs whenever data in one storage element is transferred into another. This does not necessarily change the bindings of addresses and values. The same data may reside in multiple physical locations as it does in the uniprocessor storage hierarchy, but the one nearest to the processor is the only one that the

processor can observe. If it is updated, the other hidden replicas, including the actual memory location, must eventually be updated. Copying data binds a new set of addresses to the same set of values. Generally, this will cause data transfers. Once the copy has been made, the two sets of bindings are completely independent (unlike the implicit replication that occurs within the storage hierarchy), so updates to one set of addresses do not affect the other. *Communication* between processes occurs when data written by one process is read by another. This may cause a data transfer within the machine, either on the write or the read, or the data transfer may occur for other reasons. Communication may involve establishing a new binding or not doing so, depending on the particular communication abstraction.

In general, replication avoids unnecessary communication; that is, transferring data to a consumer that was not produced since the data was previously accessed. The ability to perform replication automatically at a given level of the communication architecture depends very strongly on the naming and ordering properties of the layer. Moreover, replication is not a panacea—it too requires data transfers. It is disadvantageous to replicate data that is not going to be used. We will see that replication plays an important role throughout parallel computer architecture.

1.3.4 Performance

In defining the set of operations for communication and cooperation, the data types, and the addressing modes, the communication abstraction specifies how shared objects are named, what ordering properties are preserved, and how synchronization is performed. However, the performance characteristics of the available primitives determine how they are actually used. Programmers and compiler writers will avoid costly operations where possible. In evaluating architectural trade-offs, the decision between feasible alternatives ultimately rests upon the performance they deliver. Thus, to complete an introduction to the fundamental issues of parallel computer architecture, we need a framework for understanding performance at many levels of design.

Fundamentally, there are three important metrics: *latency*, the time taken for an operation; *bandwidth*, the rate at which operations are performed; and *cost*, the impact these operations have on the execution time of the program. In a simple world where processors do only one thing at a time, these metrics are directly related—the bandwidth (operations per second) is the reciprocal of the latency (seconds per operation), and the cost is simply the latency times the number of operations performed. However, modern computer systems do many different operations at once, and the relationship between these performance metrics is much more complex. Consider the following basic example.

EXAMPLE 1.2 Suppose a component can perform a specific operation in 100 ns. Clearly, it can support a bandwidth of 10 million operations per second. However, if the component is pipelined internally as 10 equal stages, it is able to provide a peak bandwidth of 100 million operations per second. The rate at which operations can be initiated is determined by how long the slowest stage is occupied, 10 ns, rather than by the latency of an individual operation. The bandwidth delivered on

an application depends on how frequently it initiates the operations. If the application starts an operation every 200 ns, the delivered bandwidth is 5 million operations per second, regardless of how the component is pipelined. Of course, usage of resources is usually bursty, so pipelining can be advantageous even when the average initiation rate is low. If the application performed 100 million operations on this component, what is the range of cost of these operations?

Answer Taking the operation count times the operation latency would give an upper bound of 10 seconds. Taking the operation count divided by the peak rate gives a lower bound of 1 second. The former is accurate if the program waited for each operation to complete before continuing. The latter assumes that the operations are completely overlapped with other useful work, so the cost is simply the cost to initiate the operation. Suppose that on average the program can do 50 ns of useful work after each operation issued to the component before it depends on the operations result. Then the cost to the application is 50 ns per operation—the 10 ns to issue the operation and the 40 ns spent waiting for it to complete—so the total cost is 5 seconds. ■

Since the unique property of parallel computer architecture is communication, the operations that we are concerned with most often are data transfers. The performance of these operations can be understood as a generalization of our basic pipeline example.

Data Transfer Time

The time for a data transfer operation is generally described by a linear model:

$$\text{Transfer Time}(n) = T_0 + \frac{n}{B} \tag{1.3}$$

where n is the amount of data (e.g., number of bytes), B is the transfer rate of the component moving the data in compatible units (e.g., bytes per second), and the constant term, T_0, is the start-up cost. This is a very convenient model, and it is used to describe a diverse collection of operations, including messages, memory accesses, bus transactions, and vector operations. For message passing, the start-up cost can be thought of as the time for the first bit to get to the destination. For memory operations, it is essentially the access time. For bus transactions, it reflects the bus arbitration and command phases. For any sort of pipelined operation, including pipelined instruction processing or vector operations, it is the time to fill the pipeline.

Using this simple model, it is clear that the bandwidth of a data transfer operation depends on the transfer size. As the transfer size increases, it approaches the asymptotic rate of B, which is sometimes referred to as r_∞. How quickly it approaches this rate depends on the start-up cost. It is easily shown that the size at which half of the peak bandwidth is obtained, the *half-power point*, is given by

$$n_{\frac{1}{2}} = \frac{T_0}{B} \tag{1.4}$$

Unfortunately, this linear model does not give any indication of when the next such operation can be initiated, nor does it indicate whether other useful work can be performed during the transfer. These other factors depend on how the transfer is performed.

Overhead and Occupancy

The data transfer in which we are most interested is the one that occurs across the network in parallel machines. It is initiated by the processor through the communication assist. The essential components of this operation can be described by the following simple model:

$$\text{Communication Time}(n) = \text{Overhead} + \text{Occupancy} + \text{Network Delay} \qquad (1.5)$$

The *overhead* is the time the processor spends initiating the transfer. This may be a fixed cost, if the processor simply has to tell the communication assist to start, or it may be linear in n, if the processor has to copy the data into the assist. The key point is that this is time the processor is busy with the communication event; it cannot do other useful work or initiate other communication during this time. The remaining portion of the communication time is considered the *network latency*; it is the part that can be hidden by other processor operations.

The *occupancy* is the time it takes for the data to pass through the slowest component on the communication path. For example, each link that is traversed in the network will be occupied for time n/B, where B is the bandwidth of the link. The data will occupy other resources, including buffers, switches, and the communication assist. Often the communication assist is the bottleneck that determines the occupancy. The occupancy limits how frequently communication operations can be initiated. The next data transfer will have to wait until the critical resource is no longer occupied before it can use that same resource. If there is buffering between the processor and the bottleneck, the processor may be able to issue a burst of transfers at a frequency greater than $1/\text{Occupancy}$; however, once this buffer is full, the processor must slow to the rate set by the occupancy. A new transfer can start only when an older one finishes.

The remaining communication time is lumped into the *network delay*, which includes the time for a bit to be routed across the actual network as well as many other factors, such as the time to get through the communication assists. From the processor's viewpoint, the specific hardware components contributing to network delay are indistinguishable. What affects the processor is how long it must wait before it can use the result of a communication event, how much of this time it can use for other activities, and how frequently it can communicate data. Of course, the task of designing the network and its interfaces is very concerned with the specific components and their contribution to the aspects of performance that the processor observes.

In the simple case where the processor issues a request and waits for the response, the breakdown of the communication time into its three components is immaterial.

All that matters is the total round-trip time. However, in the case where multiple operations are issued in a pipelined fashion, each of the components has a specific influence on the delivered performance.

Indeed, every individual component along the communication path can be described by its delay and its occupancy. The network delay is simply the sum of the delays along the path. The network occupancy is the maximum of the occupancies along the path. For interconnection networks, an additional factor arises because many transfers can take place simultaneously. If two of these transfers attempt to use the same resource at once (e.g., they use the same wire at the same time), one must wait. This *contention* for resources increases the average communication time. From the processor's viewpoint, contention appears as increased occupancy. Some resource in the system is occupied for a time determined by the collection of transfers across it.

Equation 1.5 is a very general model. It can be used to describe data transfers in many places in modern, highly pipelined computer systems. As one example, consider the time to move a block between cache and memory on a miss. The cache controller spends a period of time inspecting the tag to determine that it is not a hit and then starting the transfer; this is the overhead. The occupancy is the block size divided by the bus bandwidth, unless there is some slower component in the system. The delay includes the normal time to arbitrate and gain access to the bus plus the time spent delivering data into the memory. Additional time spent waiting to gain access to the bus or waiting for the memory bank cycle to complete is due to contention. A second obvious example is the time to transfer a message from one processor to another.

Communication Cost

The bottom line is, of course, the time a program spends performing communication. A useful model connecting the program characteristics to the hardware performance is given by the following:

$$\text{Communication Cost} = \text{Frequency} \times (\text{Communication Time} - \text{Overlap}) \quad (1.6)$$

The *frequency of communication*, defined as the number of communication operations per unit of work in the program, depends on many programming factors (as we will see in Chapters 2 and 3) and many hardware design factors. In particular, hardware may limit the transfer size and thereby determine the minimum number of messages. It may automatically replicate data or migrate it to where it is used. However, a certain amount of communication is inherent to parallel execution since data must be shared and processors must coordinate their work. In general, for a machine to support programs with a high communication frequency, the other parts of the communication cost equation must be small—low overhead, low network delay, and small occupancy. The attention paid to communication costs essentially determines which programming models a machine can realize efficiently and what portion of the application space it can support. Any parallel computer with good computational performance can support programs that communicate infrequently, but as the

frequency or volume of communication increase, greater stress is placed on the communication architecture.

The *overlap* is the portion of the communication operation that is performed concurrently with other useful work, including computation or other communication. This reduction of the effective cost is possible because much of the communication time involves work done by components of the system other than the processor, such as the communication assist, the bus, the network, or the remote processor or memory. Overlapping communication with other work is a form of small-scale parallelism, as is the instruction-level parallelism exploited by fast microprocessors. In effect, we may invest some of the available parallelism in a program to hide the actual cost of communication.

1.3.5 Summary

The issues of naming, operation set, and ordering apply at each level of abstraction in a parallel architecture, not just the programming model. In general, a level of translation or run-time software may intervene between the programming model and the communication abstraction, and beneath this abstraction are key hardware abstractions. At any level, communication and replication are deeply related. Whenever two processes access the same data, the data either needs to be communicated between the two or replicated so each can access a copy of it. The ability to have the same name refer to two distinct physical locations in a meaningful manner at a given level of abstraction depends on the position adopted on naming and ordering at that level. Wherever data movement is involved, we need to understand its performance characteristics in terms of latency and bandwidth and, furthermore, how these are influenced by overhead and occupancy. As architects, we need to design against the frequency and type of operations that occur at the communication abstraction, understanding that trade-offs occur across this boundary, involving what is supported directly in hardware and what is supported in software. The position adopted on naming, operation set, and ordering at each of these levels has a qualitative impact on these trade-offs, as we will see throughout the book.

1.4 CONCLUDING REMARKS

Parallel computer architecture forms an important thread in the evolution of computer architecture, rooted essentially in the beginnings of computing. For much of this history it takes on a novel, even exotic role as the avenue for advancement over and beyond what the base technology can provide. Parallel computer designs have demonstrated a rich diversity of structure, usually motivated by specific higher-level parallel programming models. However, the dominant technological forces of the VLSI generation have pushed parallelism increasingly into the mainstream, making parallel architecture almost ubiquitous. All modern microprocessors are highly parallel internally, executing several bit-parallel instructions in every cycle and even reordering instructions within the limits of inherent dependences to mitigate the

costs of communication with hardware components external to the processor itself. These microprocessors have become the performance and price-performance leaders of the computer industry. From the most powerful supercomputers to departmental servers to the desktop, we see systems constructed by utilizing multiples of such processors integrated into a communications fabric. This technological focus, and increasing maturity of compiler technology, has brought about a dramatic convergence in the structural organization of modern parallel machines. The key architectural issue is how communication is integrated into the memory and I/O systems that form the remainder of the computational node. This communications architecture reveals itself functionally in terms of what can be named at the hardware level, what ordering guarantees are provided, and how synchronization operations are performed whereas, from a performance point of view, we must understand the inherent latency and bandwidth of the available communication operations. Thus, modern parallel computer architecture carries with it a strong engineering component, amenable to quantitative analysis of cost and performance trade-offs.

This book presents the conceptual foundations as well as the engineering issues of parallel computer architecture across a broad range of potential scales of design, all of which have an important role in computing today and in the future. Computer systems, whether parallel or sequential, are designed against the requirements and characteristics of intended workloads. For conventional computers, we assume that most practitioners in the field have a good understanding of what sequential programs look like, how they are compiled, and what level of optimization is reasonable to assume that the programmer has performed. Thus, we are comfortable taking popular sequential programs, compiling them for a target architecture, and drawing conclusions from running the programs or evaluating execution traces. When we attempt to improve performance through architectural enhancements, we assume that the program is reasonably good in the first place.

The situation with parallel computers is quite different. Much less general understanding exists about the process of parallel programming, and programmer and compiler optimizations have a wider scope, which can greatly affect the program characteristics exhibited at the machine level.

Chapter 2 provides an overview of parallel programs—what they look like and how they are constructed. Chapter 3 explains the issues that must be addressed by the programmer and compiler to construct a “good” parallel program, that is, one that is effective enough in using multiple processors to form a reasonable basis for architectural evaluation. Ultimately, we design parallel computers against the program characteristics at the machine level, so the goal of Chapter 3 is to draw a connection between what appears in the program text and how the machine spends its time. In effect, Chapters 2 and 3 take us from a general understanding of issues at the application level to a specific understanding of the character and frequency of operations at the communication abstraction level.

Chapter 4 establishes a framework for workload-driven evaluation of parallel computer designs. Two related scenarios are addressed. First, for a parallel machine that has already been built, we need a sound method of evaluating its performance. This proceeds by first determining the capability of individual aspects of the

machine in isolation and then measuring how well they perform collectively. The understanding of application characteristics is important to ensure that the workload run on the machine stresses the various aspects of interest. Second, we need a process for evaluating hypothetical architectural advancements. New ideas for which no machine exists need to be evaluated through simulations, which imposes severe restrictions on what can reasonably be executed. Again, an understanding of application characteristics and how they scale with problem and machine size is crucial to navigating the design space.

Chapters 5 and 6 study in detail the design of symmetric multiprocessors with a shared physical address space. Going deeply into the small-scale case before examining scalable designs is important for several reasons. First, small-scale multiprocessors are the most prevalent form of parallel architecture; they are likely to be the form most students are exposed to, most software developers are targeting, and most professional designers are dealing with. Second, the issues that arise in the small scale are indicative of what is critical in the large scale, but the solutions are often simpler and easier to grasp. Thus, these chapters provide a study in the small of what the following five chapters address in the large. Third, the small-scale multiprocessor design is a fundamental building block for the larger-scale machines. The available options for interfacing a scalable interconnect with a processor-memory node are largely circumscribed by the processor, cache, and memory structure of the small-scale machines. Finally, the solutions to key design problems in the small-scale case are elegant in their own right.

The fundamental building block for the designs in Chapters 5 and 6 is the shared bus between processors and memory. The basic problem that we need to solve is to keep the contents of the caches coherent and the view of memory provided to the processors consistent. A bus is a powerful mechanism. It provides any-to-any communication through a single set of wires; moreover, it can serve as a broadcast medium, since there is only one set of wires, and even provide global status via wired-OR signals. The properties of bus transactions are exploited in designing extensions of conventional cache controllers that solve the coherence problem. Chapter 5 presents the fundamental techniques for bus-based cache coherence at the logical level and presents the basic design alternatives. These design alternatives provide an illustration of how workload-driven evaluation can be brought to bear in making design decisions. Finally, Chapter 5 examines the parallel programming issues of the earlier chapters in terms of the aspects of machine design that influence software level, especially with regard to cache effects on sharing patterns and the design of robust synchronization routines. Chapter 6 focuses on the organizational structure and machine implementation of bus-based cache coherence. It examines a variety of more advanced designs that seek to reduce latency and increase bandwidth while preserving a consistent view of memory.

Chapters 7 through 11 form a closely interlocking study of the design of scalable parallel architectures. Chapter 7 makes the conceptual step from a bus transaction as a building block for higher-level abstractions to a network transaction as a building block. To cement this understanding, the communication abstractions that we have

surveyed in this introductory chapter are constructed from primitive network transactions. Then the chapter studies the design of the node-to-network interface in depth using a spectrum of case studies.

Chapters 8 and 9 go deeply into the design of scalable machines supporting a shared address space, both a shared physical address space and a shared virtual address space upon independent physical address spaces. The central issue is automatic replication of data while preserving a consistent view of memory and avoiding performance bottlenecks. The study of a global physical address space emphasizes hardware organizations that provide efficient, fine-grained sharing. The study of a global virtual address space provides an understanding of a minimal degree of hardware support required for most workloads.

Chapter 10 takes up the question of the design of the scalable network itself. As with processors, caches, and memory systems, the network design space has several dimensions, and often a design decision involves interactions along these dimensions. The chapter lays out the fundamental design issues for scalable interconnects, illustrates the common design choices, and evaluates them relative to the requirements established in Chapters 8 and 9. Chapter 11 draws together the material from the previous four chapters in the context of techniques for latency tolerance, including bulk transfer, write behind, and read ahead across the spectrum of communication abstractions. Finally, Chapter 12 looks at the overall concepts of the book in light of technological, application, and economic trends and forecasts the key ongoing developments in the field of parallel computer architecture.

1.5 HISTORICAL REFERENCES

Parallel computer architecture has a long, rich, and varied history that is deeply interwoven with advances in the underlying processor, memory, and network technologies. The first blossoming of parallel architectures occurs around 1960. This is a point where transistors have replaced tubes and other complicated and constraining logic technologies. Processors are smaller and more manageable. A relatively cheap, inexpensive storage technology exists (core memory), and computer architectures are settling down into meaningful "families."

Small-scale shared memory multiprocessors took on an important commercial role at this point with the inception of what we call mainframes today, including the Burroughs B5000 (Lonergan and King 1961) and D825 (Anderson et al. 1962) and the IBM System 360 models 65 and 67 (Padegs 1981). Support for multiprocessor configurations was one of the key extensions in the evolution of the 360 architecture to System 370. These included atomic memory operations and interprocessor interrupts. In the scientific computing area, shared memory multiprocessors were also common. The CDC 6600 provided an asymmetric shared memory organization to connect multiple peripheral processors with the central processor, and a dual CPU configuration of this machine was produced. The origins of message-passing machines come about in the RW400, introduced in 1960 (Porter 1960). Data parallel machines also emerged, with the design of the Solomon computer (Ball et al. 1962; Slotnick, Borck, and McReynolds 1962).

Through the late 1960s, tremendous innovation occurred in the use of parallelism within the processor using pipelining and replication of function units to obtain a far greater range of performance within a family than could be obtained by simply increasing the clock rate. It was argued that these efforts were reaching a point of diminishing returns, so the University of Illinois and Burroughs undertook a major research project to design and build a 64-processor SIMD machine, called Illiac IV (Bouknight et al. 1972), based on the earlier Solomon work (and in spite of Amdahl's arguments to the contrary [Amdahl 1967]). This project was very ambitious, involving research in the basic hardware technologies, architecture, I/O devices, operating systems, programming languages, and applications. By the time a scaled-down, 16-processor system was working in 1975, the computer industry had undergone massive structural change.

First, the concept of storage as a simple linear array of moderately slow physical devices had been revolutionized, beginning with the idea of virtual memory and then with the concept of caching. Work on Multics and its predecessors (e.g., Atlas and CTSS) separated the concept of the user address space from the physical memory of the machine. This required maintaining a short list of recent translations, a translation lookaside buffer (TLB), in order to obtain reasonable performance. Maurice Wilkes, the designer of EDSAC, saw this as a powerful technique for organizing the addressable storage itself, giving rise to what we now call the cache. This proved an interesting example of locality triumphing over parallelism. The introduction of caches into the 360/85 yielded higher performance than the 360/91, which had a faster clock rate, faster memory, and elaborate pipelined instruction execution with dynamic scheduling. The use of caches was commercialized in the IBM 360/185, but this raised a serious difficulty for the I/O controllers as well as the additional processors. If addresses were cached and therefore not bound to a particular memory location, how was an access from another processor or controller to locate the valid data? One solution was to maintain a directory of the location of each cache line, an idea that has regained importance in recent years.

Second, storage technology itself underwent a revolution with semiconductor memories replacing core memories. Initially, this technology was most applicable to small cache memories. Other machines, such as the CDC 7600, simply provided a separate, small, fast, explicitly addressed memory. Third, integrated circuits took hold. The combined result was that uniprocessor systems enjoyed a dramatic advance in performance, which mitigated much of the added value of parallelism in the Illiac IV system, with its inferior technological and architectural base. Pipelined vector processing in the CDC STAR-100 addressed the class of numerical computations that Illiac was intended to solve but eliminated the difficult data movement operations. The final straw was the introduction of the CRAY-1 system, with an astounding 80-MHz clock rate owing to exquisite circuit design and the use of what we now call a RISC instruction set, augmented with vector operations using vector registers and offering high peak rate with very low start-up cost. The use of simple vector processing coupled with fast, expensive ECL circuits was to dominate high-performance computing for the next 15 years.

A fourth dramatic change occurred in the early 1970s, however, with the introduction of microprocessors. Although the performance of the early microprocessors was quite low, the improvements were dramatic as bit-slice designs gave way to 4-bit, 8-bit, 16-bit, and full-word designs. The potential of this technology motivated a major research effort at Carnegie-Mellon University to design a large shared memory multiprocessor using the LSI-11 version of the popular PDP-11 minicomputer. This project went through two phases. The first, called C.mmp, connected 16 processors through a specially designed circuit-switched crossbar to a collection of memories and I/O devices, much like the dancehall design in Figure 1.15 (Wulf, Levin, and Person 1975). The second, CM*, sought to build a 100-processor system by connecting 14-node clusters with local memory through a packet-switched network in a NUMA configuration (Swan, Fuller, and Siewiorek 1977; Swan et al. 1977), as in Figure 1.19.

This trend toward systems constructed from many small microprocessors literally exploded in the early to mid-1980s, resulting in the emergence of several disparate factions. On the shared memory side, it was observed that a confluence of caches and the properties of buses made modest multiprocessors very attractive. Buses have limited bandwidth but are a broadcast medium. Caches filter bandwidth and provide an intermediary between the processor and the memory system. Research at the University of California, Berkeley and elsewhere (Goodman 1983; Hill et al. 1986) introduced extensions of the basic bus protocol that allowed the caches to maintain a consistent state. This direction was picked up by several small companies, including Synapse (Nestle and Inselberg 1985), Sequent (Rodgers 1985), Encore (Bell 1985; Schanin 1986), Flex (Matelan 1985), and others, as the 32-bit microprocessor made its debut and the vast personal computer industry took off. A decade later, this general approach dominated the server and high-end workstation market and took hold in the PC servers and the desktop. The approach experienced a temporary setback as very fast RISC microprocessors took away the performance edge of multiple slower processors. Although the RISC micros were well suited to multiprocessor design, their bandwidth demands severely limited scaling until a new generation of shared bus designs emerged in the early 1990s.

Simultaneously, the message-passing direction took off with two major research efforts. At CalTech, a project was started to construct a 64-processor system using i8086/8087 microprocessors assembled in a hypercube configuration (Seitz 1985; Athas and Seitz 1988). From this baseline, several other designs were pursued at CalTech and JPL (Fox et al. 1988), and at least two companies pushed the approach into commercialization—Intel, with the iPSC series, and Ametek. A somewhat more aggressive approach was widely promoted by the INMOS Corporation in England in the form of the Transputer, which integrated four communication channels directly onto the microprocessor. This approach was adopted by nCUBE, with a series of very large-scale message-passing machines. Intel carried the commodity processor approach forward, replacing the i80386 with the faster i860, then replacing the network with a fast grid-based interconnect in the Delta and adding dedicated message processors in the Paragon. Meiko moved away from the Transputer to the i860 in

their computing surface. IBM also investigated an i860-based design in Vulcan before obtaining commercial success with the SP family, essentially a cluster of RS6000 workstations.

Data parallel systems also took off in the early 1980s, after a period of relative quiet. These included Batcher's MPP system for image processing developed by Goodyear and the Connection Machine promoted by Hillis for AI applications (Hillis 1985). The key enhancement was the provision of a general-purpose interconnect for problems demanding other than simple grid-based communication. These ideas saw commercialization with the emergence of Thinking Machines Corporation, first with the CM-1, which was close to Hillis's original conceptions, and then with the CM-2, which incorporated a large number of bit-parallel floating-point units. In addition, MasPar and Wavetracer carried the bit-serial or slightly wider organization forward in cost-effective systems.

A more formal development of highly regular parallel systems emerged in the early 1980s as systolic arrays, generally under the assumption that a large number of very simple processing elements would fit on a single chip. It was envisioned that these arrays would provide cheap, high-performance, special-purpose add-ons to conventional computer systems. To some extent, these ideas have been employed in programming data parallel machines. The iWARP project at CMU produced a more general, smaller-scale building block that has been developed further in conjunction with Intel. These ideas have also found their way into fast graphics, compression, and rendering chips.

The technological possibilities of the VLSI revolution also prompted the investigation of more radical architectural concepts, including dataflow architectures (Dennis 1980; Gurd, Kerkham, and Watson 1985; Papadopoulos and Culler 1990; Arvind and Culler 1986), which integrated the network very closely with the instruction scheduling mechanism of the processor. It was argued that very fast dynamic scheduling throughout the machine would hide the long communication latency and synchronization costs of a large machine and thereby vastly simplify programming. The evolution of these ideas tended to converge with the evolution of message-passing architectures in the form of message-driven computation (Dally, Keen, and Noakes 1993).

Large-scale shared memory designs took off as well. IBM pursued a high-profile research effort with the RP-3 (Pfister et al. 1985), which sought to connect a large number of early RISC processors (the 801) through a butterfly network. This was based on the NYU Ultracomputer work (Gottlieb et al. 1983), which was particularly novel for its use of combining operations. BBN developed two large-scale designs, the BBN Butterfly using Motorola 68000 processors and the TC2000 (Bolt Beranek and Newman 1989) using the 88100s. These efforts prompted a very broad investigation of the possibility of providing cache-coherent shared memory in a scalable setting. The DASH project at Stanford University sought to provide a fully cache-coherent distributed shared memory by maintaining a directory containing the disposition of every cache block (Lenoski et al. 1993; Lenoski et al. 1992). SCI represented an effort to standardize an interconnect and cache coherence protocol

(IEEE 1993). The Alewife project at MIT sought to minimize the hardware support for shared memory (Agarwal et al. 1995), which was pushed further by researchers at the University of Wisconsin (Wood et al. 1993). The Kendall Square Research KSR1 (Frank, Burkhardt, and Rothnie 1993; Saavedra, Gains, and Carlton 1993) went even further and allowed the home location of data in memory to migrate. Alternatively, the Denelcor HEP attempted to hide the cost of remote memory latency by interleaving many independent threads on each processor.

The 1990s have exhibited the beginnings of a dramatic convergence among these various factions. This convergence is driven by many factors. One is clearly that all of the approaches have common requirements. They all require a fast, high-quality interconnect. They all profit from avoiding latency where possible and reducing the absolute latency when it does occur. They all benefit from hiding as much of the communication cost as possible. They all must support various forms of synchronization. We have seen the shared memory work explicitly seek to better integrate message passing in Alewife (Agarwal et al. 1995) and FLASH (Kuskin et al. 1994) to obtain better performance where the regularity of the application can provide large transfers. We have seen data parallel designs incorporate complete commodity processors in the CM-5 (Leiserson et al. 1996), allowing very simple processing of messages at the user level, which provides much better efficiency for message-driven computing and shared memory (von Eicken et al. 1992; Spertus et al. 1993). There remains the additional support for fast global synchronization. We have seen fast global synchronization, message queues, and latency-hiding techniques developed in a NUMA shared memory context in the CRAY T3D (Kessler and Schwarzmeier 1993; Koeninger, Furtney, and Walker 1994), and the message-passing support in the Meiko CS-2 (Barton, Crownie, and McLaren 1994; Homewood and McLaren 1993) provides direct virtual-memory-to-virtual-memory transfers within the user address space. The new element that continues to separate the factions is the use of complete commodity workstation nodes, as in the SP-1, SP-2, and various workstation clusters using merging high-bandwidth networks (Anderson, Culler, and Patterson 1995; Kung et al. 1989; Pfister 1995). The costs of weaker integration into the memory system, imperfect network reliability, and general-purpose system requirements have tended to keep these systems more closely aligned with traditional message passing, although the future developments are far from clear.

1.6 EXERCISES

1.1 Compute the annual growth rate in number of transistors, die size, and clock rate by fitting an exponential to the technology leaders using the data in Table 1.1. Obtain more recent data from the Web, and see how well these trends have held.

1.2 Compute the annual performance growth rates for each of the benchmarks shown in Table 1.2. Comment on the differences that you observe.

Table 1.1 Basic Parameters for Several Microprocessors

Name	Year	Die (mm^2)	Total Transistors	Clock (MHz)
i4004	1971	9	2,300	0.5
i8008	1972	12.25	3,500	0.8
i8080	1974	20.25	5,000	3
M6800	1974	25	5,000	1
M68000	1979	43.56	68,000	12.5
i80286	1982	64	130,000	10
M68020	1984	84.64	180,000	25
i80386	1985	90.25	275,000	16
i80486	1988	160	1,200,000	50
MIPS R3000	1988	72	125,000	33
Motorola 68040	1989	126.4	1,200,000	25
Alpha 21064	1992	233.5	1,680,000	160
Pentium 66	1993	294	3,100,000	66.7
Alpha 21066	1994	209	1,750,000	133
MIPS R10000	1994	298	5,900,000	200
Alpha 21164	1995	298.7	9,300,000	300
UltracSparc	1995	315	3,800,000	167

Table 1.2 Performance of Leading Workstations

Machine	Year	SpecInt	SpecFP	LINPACK	n = 1,000	Peak FP
Sun 4/260	1987	9	6	1.1	1.1	3.3
MIPS M/120	1988	13	10.2	2.1	4.8	6.7
MIPS M/2000	1989	18	21	3.9	7.9	10
IBM RS6000/540	1990	24	44	19	50	60
HP 9000/750	1991	51	101	24	47	66
DEC Alpha AXP	1992	80	180	30	107	150
DEC 7000/610	1993	132.6	200.1	44	156	200
AlphaServer 2100	1994	200	291	43	129	190

1.3 Generally in evaluating performance trade-offs, we evaluate the improvement in performance, or speedup, due to some enhancement. Formally,

$$\text{Speedup due to enhancement } E = \frac{\text{Time}_{\text{without } E}}{\text{Time}_{\text{with } E}} = \frac{\text{Performance}_{\text{with } E}}{\text{Performance}_{\text{without } E}}$$

In particular, we will often refer to the speedup as a function of the machine parallel (e.g., the number of processors).

Suppose you are given a program that does a fixed amount of work, and some fraction s of that work must be done sequentially. The remaining portion of the work is perfectly parallelizable on P processors. Assuming T_1 is the time taken on one processor, derive a formula for T_p, the time taken on P processors. Use this to get a formula giving an upper bound on the potential speedup on P processors. (This is a variant of what is often called Amdahl's Law [Amdahl 1967].) Explain why it is an upper bound.

1.4 Given a histogram of available parallelism such as that shown in Figure 1.7, where f_i is the fraction of cycles on an ideal machine in which i instructions issue, derive a generalization of Amdahl's Law to estimate the potential speedup on a k-issue superscalar machine. Apply your formula to the histogram data in Figure 1.7 to produce the speedup curve shown in that figure.

1.5 Locate the current TPC performance data on the Web and compare the mix of system configurations, performance, and speedups obtained on those machines with the data presented in Figure 1.4.

1.6 In message-passing models, each process is provided with a special variable or function that gives its unique number or rank among the set of processes executing a program. Most shared memory programming systems provide a fetch&inc operation, which reads the value of a location and atomically increments the location. Write a little pseudocode to show how to use fetch&add to assign each process a unique number. Can you determine the number of processes comprising a shared memory parallel program in a similar way?

1.7 To move an n-byte message along H links in an unloaded store-and-forward network takes time $H\frac{n}{W} + (H-1)R$, where W is the raw link bandwidth and R is the routing delay per hop. In a network with cut-through routing, this takes time $\frac{n}{W} + (H-1)R$. Consider an 8×8 grid consisting of 40-MB/s links and routers with 250 ns of delay. What is the minimum, maximum, and average time to move a 64-byte message through the network? A 256-byte message?

1.8 Consider a simple 2D finite difference scheme where at each step every point in the matrix is updated by a weighted average of its four neighbors, $A[i, j] = A[i, j] - w(A[i-1, j] + A[i+1, j] + A[i, j-1] + A[i, j+1])$.

All the values are 64-bit floating-point numbers. Assuming one element per processor and $1{,}024 \times 1{,}024$ elements, how much data must be communicated per step? Explain how this computation could be mapped onto 64 processors so as to minimize the data traffic. Compute how much data must be communicated per step.

1.9 Consider the simple pipelined component described in Example 1.2. Suppose that the application alternates between bursts of m independent operations on the component and phases of computation lasting T ns that do not use the component. Develop an expression describing the execution time of the program based on these parameters. Compare this with the unpipelined and fully pipelined bounds. At what points do you get the maximum discrepancy between the models? How large is it as a fraction of overall execution time?

1.10 Show that Equation 1.4 follows from Equation 1.3.

1.11 What is the x-intercept of the line in Equation 1.3?

1.12 If we consider loading a cache line from memory, the transfer time is the time to actually transmit the data across the bus. The start-up includes the time to obtain access to the bus, convey the address, access the memory, and possibly place the data in the cache before responding to the processor. However, in a modern processor with dynamic instruction scheduling, the overhead may include only the portion spent accessing the cache to detect the miss and placing the request on the bus. The memory access portion contributes to latency, which can potentially be hidden by the overlap with execution of instructions that do not depend on the result of the load.

Suppose we have a machine with a 64-bit-wide bus running at 40 MHz. It takes two bus cycles to arbitrate for the bus and present the address. The cache line size is 32 bytes and the memory access time is 100 ns. What is the latency for a read miss? What bandwidth is obtained on this transfer?

1.13 Suppose this 32-byte line is transferred to another processor and the communication architecture imposes a start-up cost of 2 μs and a data transfer bandwidth of 20 MB/s. What is the total latency of the remote operation?

1.14 If we consider sending an n-byte message to another processor, we may use the same model as in Exercise 1.12. The start-up can be thought of as the time for a zero-length message; it includes the software overhead on the two processors, the cost of accessing the network interface, and the time to actually cross the network. The transfer time is usually determined by the point along the path with the least bandwidth, that is, the bottleneck.

Suppose we have a machine with a message start-up of 100 μs and an asymptotic peak bandwidth of 80 MB/s. At what size message is half of the peak bandwidth obtained?

1.15 In some cases, Equation 1.6 can be used for estimating data transfer performance based on design parameters. In other cases, it serves as an empirical tool for fitting measurements to a line to determine the effective start-up and peak bandwidth of a portion of a system. If data undergoes a series of copies as part of a transfer (assuming that before transmitting a message the data must be copied into a buffer), the basic message time is as in Exercise 1.14, but the copy is performed at a cost of 5 cycles per 32-bit word on a 100-MHz machine. Given an equation for the expected user-level message time, how does the cost of a copy compare with a fixed cost of, say, entering the operating system?

1.16 Consider a machine running at 100 MIPS on some workload with the following mix: 50% ALU, 20% loads, 10% stores, and 20% branches. Suppose the instruction miss rate is 1%, the data miss rate is 5%, and the cache line size is 32 bytes. For the purpose of this calculation, treat a store miss as requiring two cache line transfers, one to load the newly updated line and one to replace the dirty line. If the machine provides a 250-MB/s bus, how many processors can it accommodate at 50% of peak bus bandwidth? What is the bandwidth demand of each processor?

1.17 Exercise 1.16 looks only at the sum of the average bandwidths, leaving 50% headroom on the bus to make the calculation reasonable. As the bus approaches saturation, however, it takes longer to obtain access for the bus, so it looks to the processor as if the memory system is slower. The effect is to slow down all of the processors in the system, thereby reducing their bandwidth demand. Let's try an analogous calculation from the other direction.

Assume the instruction mix and miss rate as in Exercise 1.16, but ignore the MIPS since that depends on the performance of the memory system. Assume instead that the processor runs at 100 MHz and has an ideal CPI (with a perfect memory system) of one. The unloaded cache miss penalty is 20 cycles. You can ignore the write back for stores. (As a starter, you might want to compute the MIPS rate for this new machine.) Assume that the memory system (i.e., the bus and the memory controller) is utilized throughout the miss. What is the utilization of the memory system U_1 with a single processor? From this result, estimate the number of processors that could be supported before the processor demand would exceed the available bus bandwidth.

1.18 Of course, no matter how many processors you place on the bus, they will never exceed the available bandwidth. Explain what happens to processor performance in response to bus contention. Can you formalize your observations?