1

Introduction

For over a decade, we have enjoyed explosive growth in the performa bility of computer systems. The theme of this dramatic success stor of the underlying VLSI technology, which allows clock rates to inc numbers of components to fit on a chip. The plot of this story cente architecture, which translates the raw potential of the technology formance and expanded capability of the computer system. The story's acter is parallelism. A larger volume of resources means that more c performed at once, in parallel. Parallel computer architecture is these resources so that they work well together. Computers of all nessed parallelism more and more effectively to gain performance fro nology, and the level at which parallelism is exploited continues to character is storage. The data that is operated on at an ever faste somewhere in the machine. Thus, the story of parallel processing i twined with data locality and communication. The computer architect these changing relationships to design the various levels of a comp to maximize performance and programmability within the limits impos nology and cost at any particular time.

Parallelism is a fascinating perspective from which to understand tecture because it applies at all levels of design, it interacts wi architectural concepts, and it presents a unique dependence on the u nology. In particular, the basic issues of locality, bandwidth, late zation arise at many levels of the design of parallel computer syste must be resolved in the context of real application workloads.

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

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

However, this simple definition raises many questions. How large a we talking about? How powerful are the individual processing element number be increased in a straightforward manner? How do these eleme nicate and cooperate? How is data transmitted between processors, interconnection is provided, and what operations are available to actions carried out on different processors? What are the primitive the hardware and software provide to the programmer? And finally, how does i translate into performance? In answering these questions, we will see tha moderate, and very large collections of processing elements each have imp roles to fill in modern computing. Thus, it is important to understand p machine design across the scale, from the small to the very large. Some des apply throughout the scale of parallelism; others are most germane to a p regime, such as within a chip, within a box, or on a very large machine. It say that parallel machines occupy a rich and diverse design space. This makes the area exciting, but it also means that it is important that we deve framework in which to understand the many design alternatives.

Parallel architecture is itself changing rapidly. Historically, parallel m demonstrated innovative organizational structures, often tied to specific p ming models, as architects sought to obtain the ultimate in performance c given technology. In many cases, radical organizations were justified on the that advances in the base technology would eventually run out of steam. The predictions appear to have been overstated, as logic densities and switchi have continued to improve and more modest parallelism has been employed lower levels to sustain continued improvement in processor performance. Non less, application demand for computational performance continues to outpace individual processors can deliver, and multiprocessor systems occupy an i ingly important place in mainstream computing. What has changed is the nove these parallel architectures. Even large-scale parallel machines today are the same basic components as workstations and personal computers. They are ject to the same engineering principles and cost-performance trade-offs. M to yield the utmost in performance, a parallel machine must extract the fu mance potential of its individual components. Thus, an understanding of m parallel architectures must include an in-depth treatment of engineering t not just a descriptive taxonomy of possible machine structures.

Parallel architectures will play an increasingly central role in informati ing. This view is based not so much on the assumption that individual pro performance will soon reach a plateau but rather on the estimation that t level of system design, the multiprocessor level, will become increasingly 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 tl sion of responsibilities between hardware and software in evolving pa machines. Understanding this division requires familiarity with the requi that parallel programs place on the machine and the interaction of machine and the practice of parallel programming.

The process of learning computer architecture is frequently likened to pe onion, and this analogy is even more appropriate for parallel computer arch At each level of understanding we find a complete whole with many interac facets, including the structure of the machine, the abstractions it present nology it rests upon, the software that exercises it, and the model performance. However, if we dig deeper into any of these facets, we layer of design and a new set of interactions. The holistic, multile lel computer architecture makes the field challenging to learn and present. Some sense of the layer-by-layer structure is unavoidable.

This introductory chapter presents the "outer skin" of parallel co ture. It first outlines the reasons why parallel machine design may b from desktop machines to supercomputers. It also examines the te architectural, and economic trends that have led to the current st architecture and that provide the basis for anticipating future par Section 1.1 focuses on the forces that have brought about the dram processor performance and the restructuring of the entire comput around commodity microprocessors. These forces include the insatiab demand for computing power, the continued improvements in the densit of integration in VLSI chips, and the utilization of parallelism at levels of the architecture.

Next is a quick look at the spectrum of important architectural st the field such a rich history and contribute to the modern understan machines. Within this diversity of design, a common set of design trade-offs arise, driven by the same advances in the underlying te forces are rapidly leading to a convergence in the field, which forms this book. Section 1.2 surveys traditional parallel machines, inclu ory, message passing, data parallel, systolic arrays, and dataflow, different ways that they address common architectural issues. The di the dependence of parallel architecture on the underlying technolo importantly, demonstrates the convergence that has come about with t of microprocessors.

Building on this convergence, Section 1.3 examines the fundame issues that cut across parallel machines: what can be named at the m basis for communication and coordination, what is the latency or ti perform these operations, and what is the bandwidth or overall rate can be performed? This shift from conceptual structure to performanc provides a framework for quantitative, rather than merely qualitativ lel computer architecture.

With this initial broad understanding of parallel computer archit the following chapters dig deeper into its technical substance. C delve into the structure and requirements of parallel programs to pr understanding the interaction between parallel architecture and Chapter 4 builds a framework for evaluating design decisions in term requirements and performance measurements. Chapters 5 and 6 are a study of parallel computer architecture at the limited scale empl commercial multiprocessors—from a few processors to a few tens of pr concepts and structures introduced here form the building blocks fo sive large-scale designs presented over the final five chapters.

1.1 WHY PARALLEL ARCHITECTURE

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

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

First, the performance of the highly integrated, single-chip CMOS microp sor is steadily increasing and is surpassing the larger, more expensive al Microprocessor performance has been improving at a rate of about 50% per The advantages of using small, inexpensive, low-power, mass-produced proce as the building blocks for computer systems with many processors are intu clear. However, until recently the performance of the processor best suited lel architecture was far behind that of the fastest single-processor system longer true. Although parallel machines have been built at various scales earliest days of computing, the approach is more viable today than ever because the basic processor building block is better suited to the job.

The second and perhaps more fundamental observation is that change, ev dramatic change, is the norm in computer architecture. The continuing proc change has profound implications for the study of computer architecture beca need to understand not only how things are but how they might evolve and Change is one of the key challenges in writing this book—and one of the key vations. Parallel computer architecture has matured to the point where it ne studied from a basis of engineering principles and quantitative evaluation mance and cost. These are rooted in a body of facts, measurements, and des real machines. Unfortunately, existing data and designs are necessarily froz

---

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

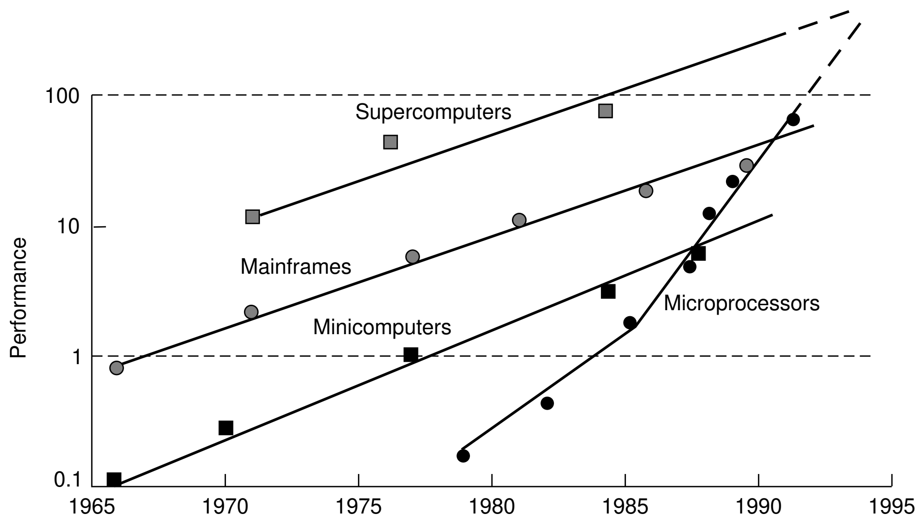

FIGURE 1.1 Performance trends over time of micros, minicomputers, mainframes, and supercomputers. Performance of microprocessors has been increasing at a rate of about 50% per year since the mid-1980s. More traditional mainframe and supercomputer performance has been increasing at a rate of roughly 25% 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 examines real machines in the form of a late 1990s technological sn to retain a clear grounding. However, the methods of evaluation unde ysis of concrete design trade-offs transcend the chronological and erence point of the book.

The late 1990s happens to be a particularly interesting snapshot in the midst of a dramatic technological realignment as the single-c sor is poised to dominate every sector of computing and as parallel hold in many areas of mainstream computing. Of course, the prevaler suggests being cautious about extrapolating into the future. The r section examines more deeply the forces and trends that are giving p tures an increasingly important role throughout the computing field parallel computing into the mainstream. It looks first at the applic increased performance and then at the underlying technological and trends that strive to meet these demands. We see that parallelis attractive as computers become more highly integrated and that it is at increasingly high levels of the design. Finally, this section clo role of parallelism in the machines at the very high end of th spectrum.

1.1.1 Application Trends

The demand for ever greater application performance is a familiar feature aspect of computing. Advances in hardware capability enable new applicatior tionality, which grows in significance and places even greater demands on th tecture. This cycle drives the tremendous ongoing design, engineering manufacturing effort underlying the sustained exponential performance incre microprocessor performance. It drives parallel architecture even harder sir lel architecture focuses on the most demanding of these applications. With annual improvement in processor performance, a parallel machine of a hundrec cessors can be viewed as providing to applications the computing power that widely available 10 years in the future, whereas a thousand processors reflec a 20-year horizon.

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

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

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

For a single, fixed problem, the performance of the machine on the probl simply the reciprocal of the time to complete the problem, so we have the f 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 ber of endeavors but is perhaps most apparent in the fields of computational and engineering. Basically, in these fields computers are used to simulate phenomena that are impossible or very costly to observe through empirical

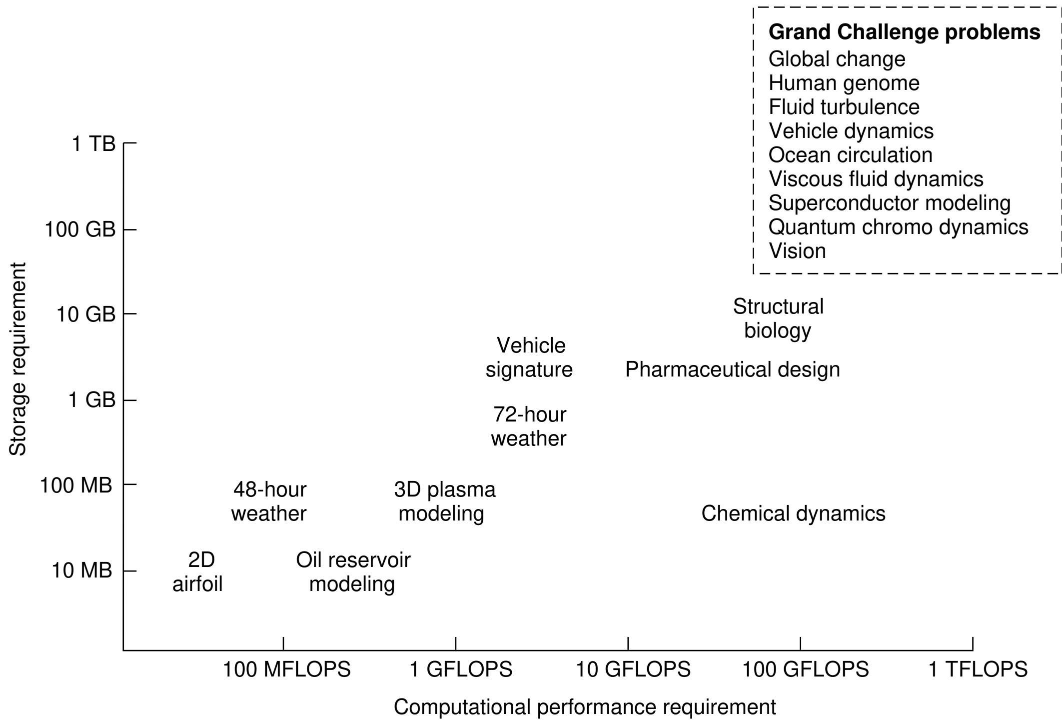

FIGURE 1.2 Grand Challenge application requirements. A collection of important scientific and engineering problems is positioned in a space defined by computational performance and storage capacity. Given the exponential growth rate of performance and capacity, both of these axes map directly to time. 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 evolution of galaxies, the atomic structure of materials, the efficie with an engine, the flow of air over surfaces of vehicles, the damag and the behavior of microscopic electronic devices. Computational mo in-depth analyses to be performed cheaply on hypothetical designs puter simulation. A direct correspondence can be drawn between level tional performance and the problems that can be studied through Figure 1.2 summarizes the 1993 findings of the Committee on Physical, cal, and Engineering Sciences of the federal Office of Science and Te (1993). It indicates the computational rate and storage capacity re number of important science and engineering problems. Even with increases in processor performance, very large parallel architectu address these problems in the near future. Some years further down grand challenges will be in view.

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

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

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

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

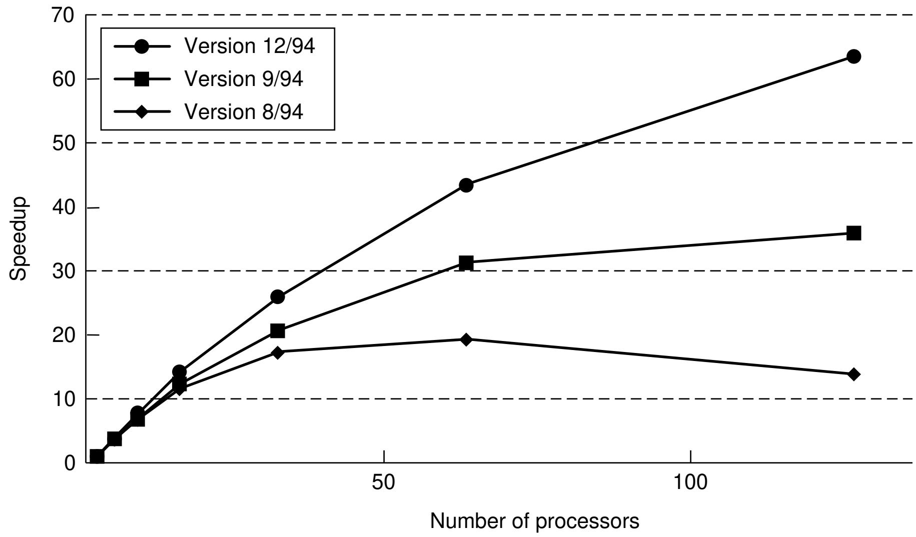

FIGURE 1.3 Speedup on three versions of a parallel program. The parallelization learning curve is illustrated by the speedup obtained on three successive versions of this molecular dynamics code on the Intel Paragon.

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

The demand for ever increasing performance is a natural consequ modeling activity. For example, in electronic CAD there is obviousl late as the number of devices on the chip increases. In addition, tl plexity of the design requires that more test vectors be used and, level functionality is incorporated into the chip, each of these t larger number of clock cycles. Furthermore, an increasing level o required because the cost of fabrication is so great. The cumulativ computational demand for the design verification of each new generat ing at an even faster rate than the performance of the microprocesso

Commercial Computing

Commercial computing has also come to rely on parallel architectur end. Although the scale of parallelism is typically not as large a puting, the use of parallelism is even more widespread. Multiproce vided the high end of the commercial computing market since the mi

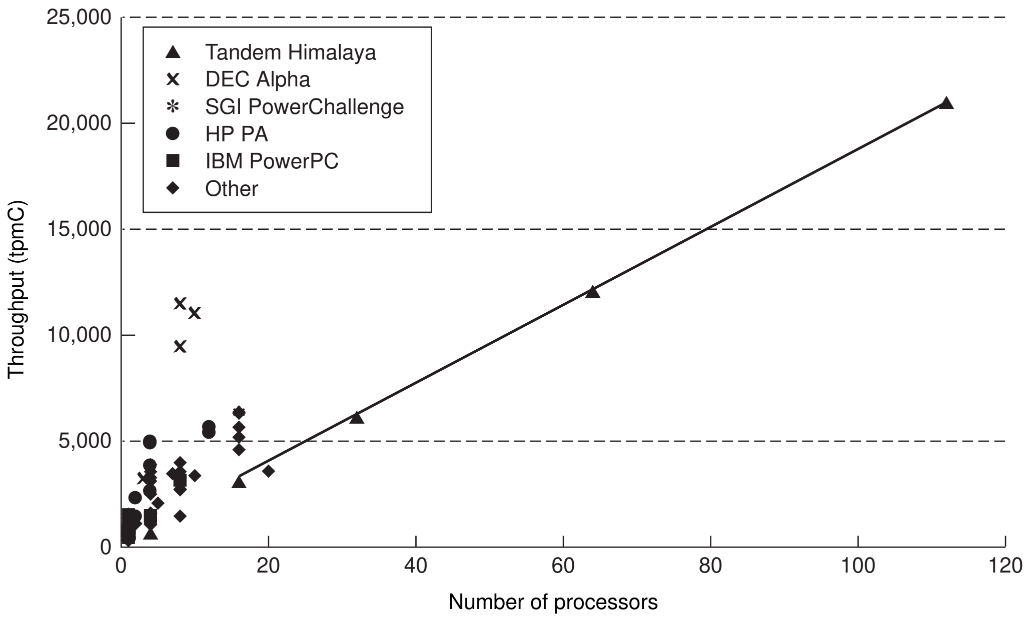

FIGURE 1.4 TPC-C throughput versus number of processors on TPC. The March 1996 TPC report documents the transaction processing performance for a wide range of systems. The figure shows the number of processors employed for all of the high-end systems, highlighting five leading vendor product lines. All 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 th business that can be supported by the system. The relationship between p mance and scale of business enterprise is clearly articulated in the on-li tion processing (OLTP) benchmarks sponsored by the Transaction Processi Performance Council (TPC). These benchmarks rate the performance of a syste 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 an transactions, including realistic features like queued transactions, abort tions, and elaborate presentation features (Gray 1991). The benchmark in explicit scaling criteria to make the problem more realistic: the size of and the number of terminals in the system increase as the tpmC (the tpm on T rating rises. Thus, a faster system must operate on a larger database and larger number of users.

Figure 1.4 shows the tpmC ratings for the collection of systems appearin edition of the TPC results (March 1996), with the achieved throughput on th cal axis and the number of processors employed in the server along the hor axis. This data includes a wide range of systems from a variety of hardware ware vendors, a few of which are highlighted here. Since the problem solve benchmark run scales with system performance, we cannot simply compare time see the effectiveness of parallelism. Instead, we use the throughpu the metric of performance in Equation 1.1. The resulting speedup i 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. Fi parallel architectures is prevalent. Essentially all of the vendors hardware or software offer multiprocessor systems that provide substantially beyond their uniprocessor product. Second, it is not parallelism that is important but modest-scale multiprocessor serv processors and even small-scale multiprocessors with two or fou

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

The transition to parallel programming, including new algorithms or atten
communication and synchronization requirements in existing algorithms,
largely taken place in the high-performance end of computing. The transiti
progress among the much broader base of commercial engineering software. T
cally, engineering and commercial applications target more modest-scale mu
cessors, which dominate the server market. In the commercial world, all
major database vendors support parallel machines for their high-end product
eral major database vendors also offer "shared-nothing" versions for large
machines and collections of workstations on a fast network, often called *clusters*. In
addition, multiprocessor machines are heavily used to improve throughput on
tiprogramming workloads. Even the desktop demonstrates a significant number
concurrent processes, with a host of active windows and daemons. Quite ofter
gle user will have tasks running on many machines within the local area net
will farm tasks out across the network. All of these trends provide a solid
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
performance can be brought into sharper focus by looking more closely a
advancements in the underlying technology and architecture. These trends s
that it may be increasingly difficult to "wait for the single processor
enough" while parallel architectures become more attractive. Moreover, the
nation shows that the critical issues in parallel computer architecture are
tally similar to those that we wrestle with in "sequential" computers, suc
the resource budget should be divided among functional units that do the
caches that exploit locality, and wires that provide communication bandwidtl

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

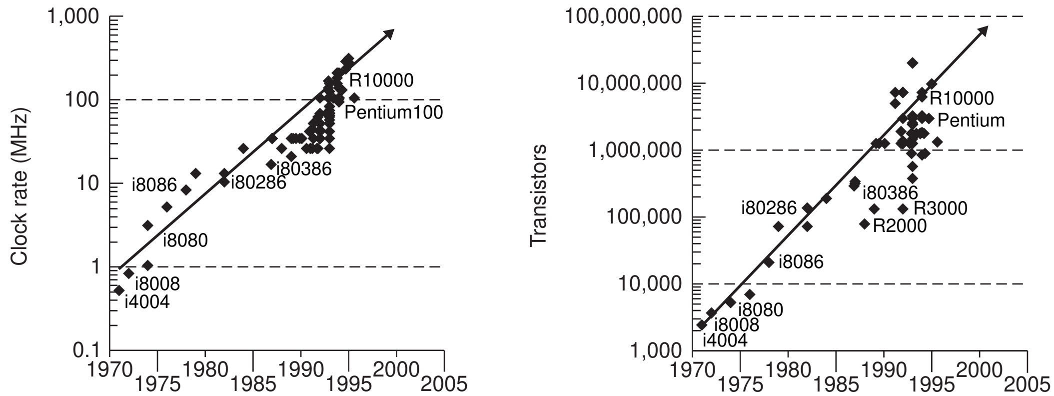

FIGURE 1.5 Improvement in logic density and clock frequency of microprocessors. Improvements in lithographic technique, process technology, circuit design, and datapath design have yielded a sustained improvement in logic density and clock rate.

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

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

---

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

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

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

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

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

1.1.3 Architectural Trends

Advances in technology determine what is possible; architecture translat potential of the technology into performance and capability. Fundamentally, ways in which a larger volume of resources (e.g., more transistors) improve mance are parallelism and locality. Moreover, these two approaches compete for the same resources. Whenever multiple operations are performed in parallel, th ber of cycles required to execute the program is reduced. However, resour required to support each of the simultaneous activities. Whenever data re are performed close to the processor, the latency of accessing deeper lev storage hierarchy is avoided and the number of cycles to execute the proc reduced. However, resources are required to provide this local storage. Ir the best performance is obtained by an intermediate strategy that devotes to exploiting a degree of parallelism and a degree of locality. Indeed, v throughout the book that parallelism and locality interact in interesting w tems of all scales, from within a chip to across a large parallel r microprocessors, the die area is divided roughly equally between cac cessing, and off-chip interconnect. Larger-scale systems may exhibit ferent split because of differences in cost and performance trade-c issues are the same.

Microprocessor Design Trends

Examining the trends in microprocessor architecture helps build in the issues we will be dealing with in parallel machines. It also i damental parallelism is to conventional computer architecture and architectural trends are leading toward multiprocessor designs. (Th processor design techniques in this book is cursory since many reade to be familiar with those techniques from traditional architecture and Patterson 1996] or the many discussions in the trade literature. unique perspective on those techniques, however, and will serve t memory.)

The history of computer architecture has traditionally been divide erations identified by the basic logic technology: tubes, transisto cuits, and VLSI. The entire period covered by the figures in this cl into the fourth, or VLSI, generation. Clearly, there has been trem tural advance over this period, but what delineates one era from the generation? The strongest delineation is the kind of parallelism th indicated in Figure 1.6.

The period up to about 1986 is dominated by advances in *bit-level parallelism*, with 4-bit microprocessors replaced by 8-bit, 16-bit, and so c width of the datapath reduces the number of cycles required to perfo operation. Once a 32-bit word size is reached in the mid-1980s, th with only partial adoption of 64-bit operation obtained a decade increases in word width will be driven by demands for improved float resentation and a larger address space rather than performance. Wit requirements growing by less than a bit per year, the demand for 12 appears to be well in the future. The early microprocessor period the benefits of the easiest form of parallelism: bit-level parallel tion. The dramatic inflection point in the microprocessor growth cu Figure 1.1 marks the arrival in 1986 of full 32-bit word operation c prevalent use of caches.

The period from the mid-1980s to the mid-1990s is dominated by ad in *instruction-level parallelism*, performing portions of several machine instru concurrently. Full-word operation meant that the basic steps in inst ing (instruction decode, integer arithmetic, and address calculatic performed in a single cycle; with caches, the instruction fetch and also be performed in a single cycle most of the time. The RISC ap strated that, with care in the instruction set design, it was straig the stages of instruction processing so that an instruction is exec

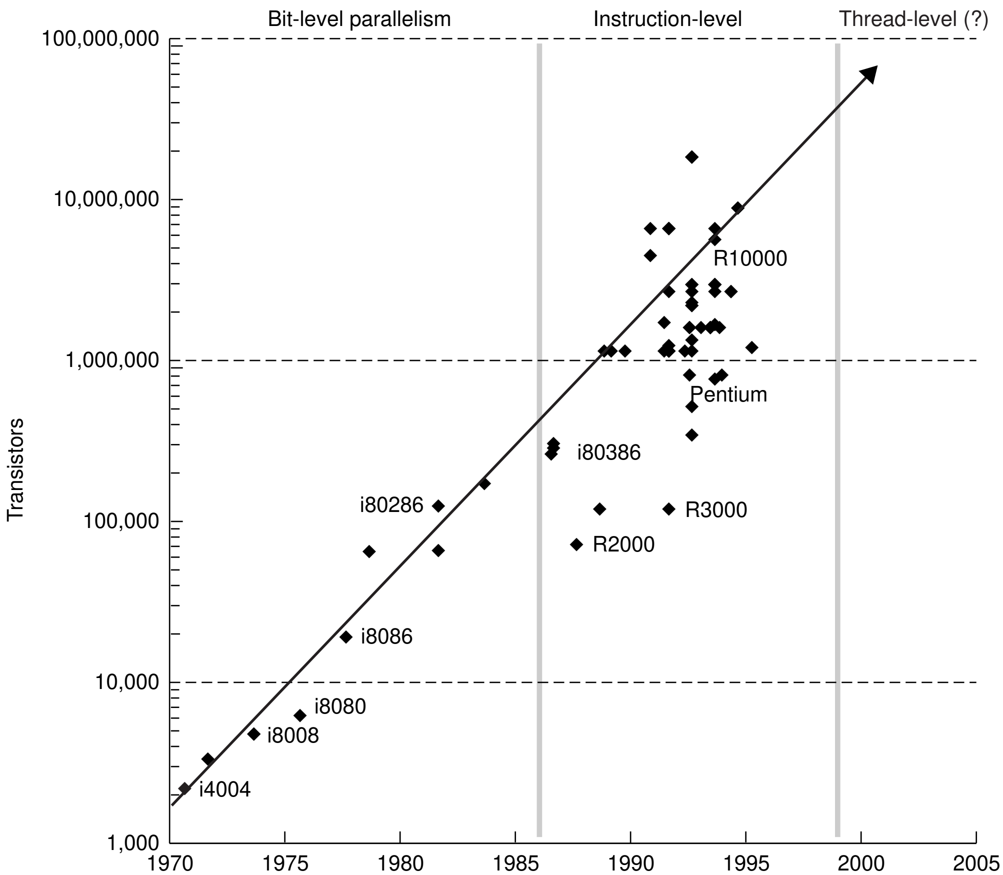

FIGURE 1.6 Number of transistors per processor chip over the last 25 years. The growth essentially follows Moore's Law, which says that the number of transistors doubles every two years. Forecasting from past trends, we can reasonably expect to be designing for a 50- to 100-million-transistor budget 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 instructi ing could be exploited across a small number of instructions. While pip instruction processing was not new, it had never before been so well suite underlying technology. In addition, advances in compiler technology made in tion pipelines more effective.

The mid-1980s microprocessor-based computers consisted of a small conste tion of chips: an integer processing unit, a floating-point unit, a cache and SRAMs for the cache data and tag storage. As chip capacity increased components were coalesced into a single chip, which reduced the cost of com cating among them. Thus, a single chip contained separate hardware for i arithmetic, memory operations, branch operations, and floating-point addition to pipelining individual instructions, it became very attra tiple instructions at a time and issue them in parallel to disti whenever possible. This form of instruction-level parallelism cam *superscalar* execution. It provided a natural way to exploit the ever in ber of available chip resources. More function units were added, mo were fetched at a time, and more instructions could be issued in ea the function units.

However, increasing the amount of instruction-level parallelism t sor can exploit is only worthwhile if the processor can be supplied and data fast enough to keep it busy. In order to satisfy the incr and data bandwidth requirement, larger and larger caches were place the processor, further consuming the ever increasing number of trans processor and cache on the same chip, the path between the two could wide to satisfy the bandwidth requirement of multiple instruction an per cycle. However, as more instructions are issued each cycle, t impact of each control transfer and each cache miss becomes more 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 instruction to execute next. Similarly, instructions that use a v memory may cause the processor to wait for the latency of a cache mi

Processor designs in the 1990s deploy a variety of complex instruc mechanisms in an effort to reduce the performance degradation re latency in "wide-issue" superscalar processors. Sophisticated bra techniques are used to avoid pipeline latency by guessing the dire flow before branches are actually resolved. Larger, more sophistica used to *avoid* the latency of cache misses. Instructions are scheduled and allowed to complete out of order so if one instruction encounte instructions can proceed ahead of it as long as they do not depend the instruction. A larger window of instructions that are waiting tained within the processor and whenever an instruction produces a r eral waiting instructions may be issued to the function units. mechanisms allow the processor to *tolerate* the latency of a cache miss or pipe dependence when it does occur. However, each of these mechanisms pla demand on chip resources and carries a very heavy design cost.

Given the expected increases in chip density, the natural questio far will instruction-level parallelism go within a single thread o point will the emphasis shift to supporting the higher levels of pa as multiple processes or multiple threads of control within a proces *thread-level parallelism*? Several research studies have sought to answer the first question, either through simulation of aggressive machine designs 1991; Horst, Harris, and Jardine 1990; Lee, Kwok, and Briggs 1991; N 1991) or through analysis of the inherent properties of programs (Bu Jouppi and Wall 1989; Johnson 1991; Smith, Johnson, and Horowitz 1991). The most complete treatment appears in Johnson's book devoted

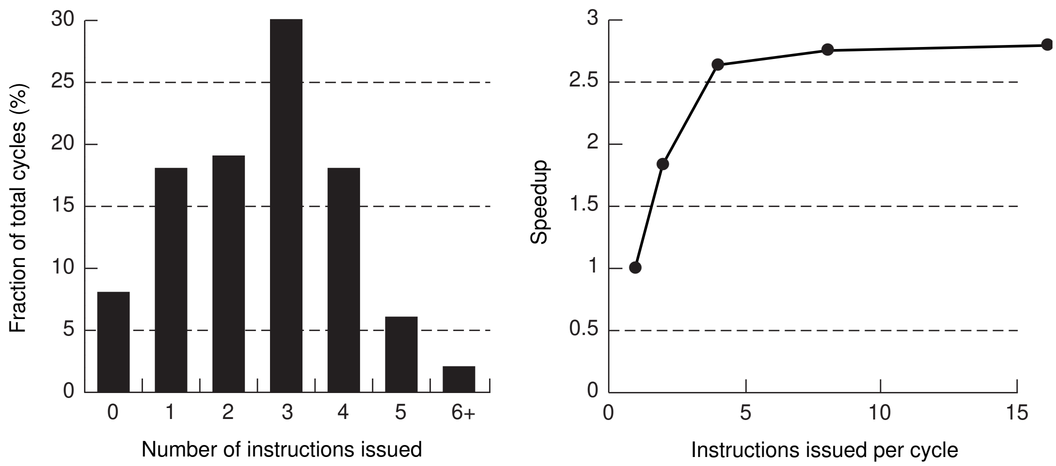

FIGURE 1.7 Distribution of potential instruction-level parallelism and estimated speedup under ideal superscalar execution. The figure shows the distribution of available instruction-level parallelism and maximum potential speedup under idealized superscalar execution, including unbounded processing 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 tw superscalar, that is, issuing two instructions per cycle, is very profitabl way offers substantial additional benefit, but wider issue widths (e.g., superscalar) provide little additional gain. The design complexity increase cally because control transfers occur roughly once in five instructions, on

To estimate the maximum potential speedup that can be obtained by issuing tiple instructions per cycle, the execution trace of a program is simulated machine with unlimited instruction fetch bandwidth, as many function units program can use, and perfect branch prediction. (The latter is easy, since correctly follows each branch.) These generous machine assumptions ensure tl instruction is held up because a function unit is busy or because the inst beyond the lookahead capability of the processor. Furthermore, to ensure t instruction is delayed because it updates a location that is used by logica instructions, storage resource dependences are removed by a technique c *renaming* Each update to a register or memory location is treated as introdu new "name," and subsequent uses of the value in the execution trace refer new name. In this way, the execution order of the program is constrained c essential data dependences; each instruction is executed as soon as its ope available. Figure 1.7 summarizes the result of this ideal machine analysis data presented by Johnson (1991). The histogram on the left shows the frac cycles in which no instruction could issue, only one instruction could issu on. Johnson's ideal machine retains realistic function unit latencies, incl misses, which accounts for the zero-issue cycles. (Other studies ig or ignore pipeline latencies and thereby obtain more optimistic est that, even with infinite machine resources, perfect branch predict renaming, no more than four instructions issue in a cycle 90% of the this distribution, we can estimate the speedup obtained at various shown in the right portion of the figure. Recent work (Lam and Wilso Breach, and Vijaykumar 1995) provides empirical evidence that to c cantly larger amounts of parallelism, multiple threads of control simultaneously. Barring some unforeseen breakthrough in instruction- ism, the leap to the next level of useful parallelism—multiple conc is increasingly compelling as chips increase in capacity.

System Design Trends

The trend toward thread- or process-level parallelism has been str puter system level for some time. Computers containing multiple st microprocessors sharing a common memory became prevalent in the mi when the 32-bit microprocessor was first introduced (Bell 1985). As Figure 1.8, which shows the number of processors available in comm processors over time, this bus-based shared memory multiprocessor maintained a substantial multiplier to the increasing performance c processors. Almost every commercial microprocessor introduced sin 1980s provides hardware support for multiprocessor configurations, a Chapter 5. Multiprocessors dominate the server and enterprise (or ma kets and have migrated to the desktop.

The early multi-microprocessor systems were introduced by small competing for a share of the minicomputer market, including Synapse Inselberg 1985), Encore (Schanin 1986), Flex (Matelan 1985), Seque 1985), and Myrias (Savage 1985). They combined 10 to 20 micropro deliver competitive throughput on time-sharing loads. With the intr 32-bit Intel i80386 as the base processor, these systems obtained mercial success, especially in transaction processing. However, t mance advance of RISC microprocessors, exploiting instruction-leve sapped the CISC multiprocessor momentum in the late 1980s and all bu the minicomputer. Shortly thereafter, several large companies began multiprocessor systems, especially as servers and mainframe replac designs highlight the critical role of bandwidth. In most of thes designs, all the processors plug into a common bus. Since a bus ha width, as the processors become faster, a smaller number can be su bus. The early 1990s brought a dramatic advance in the shared memory ogy, including faster electrical signaling, wider datapaths, pipeli multiple paths. Each of these provided greater bandwidth, growing design experience, as indicated in Figure 1.9. This increase in ba the multiprocessor designs to ramp back up to the 10-to-20 range and tracking the microprocessor advances (Alexander et al. 1994; Ceklе

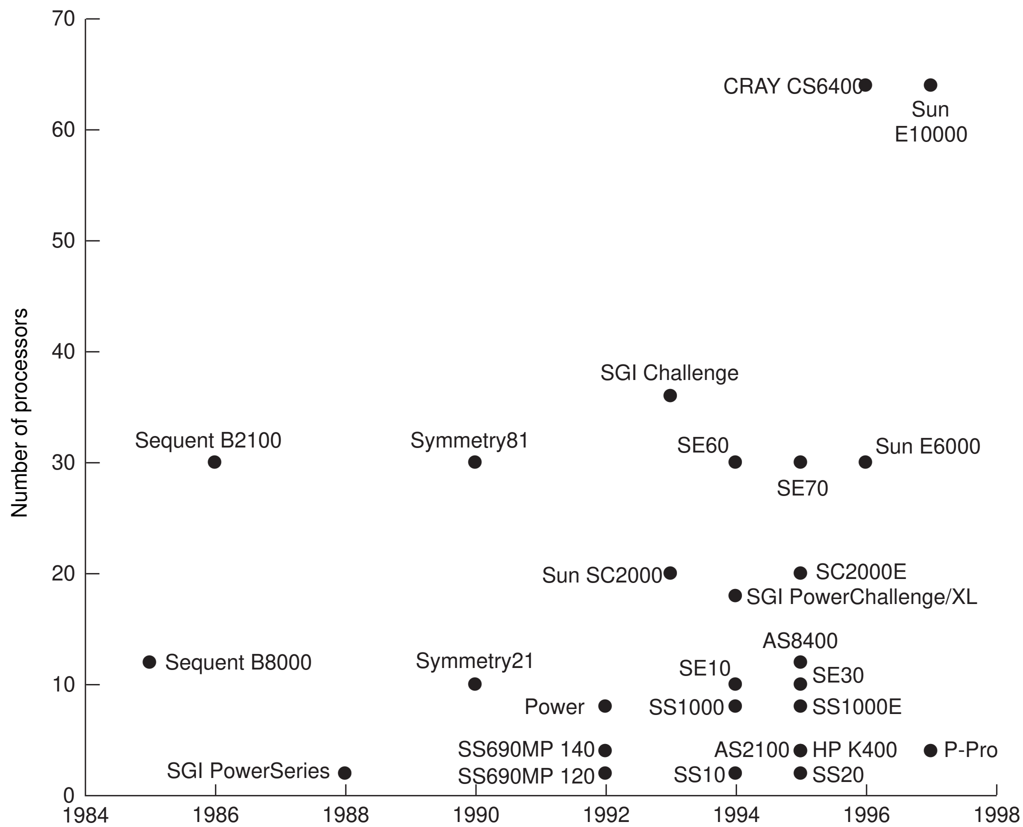

FIGURE 1.8 Number of processors in fully configured commercial bus-based shared memory multiprocessors. After an initial era of 10- to 20-way shared memory processors based on slow CISC microprocessors, companies such as Sun, HP, DEC, SGI, IBM, and CRI began producing sizable RISC-based SMPs, as did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

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

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

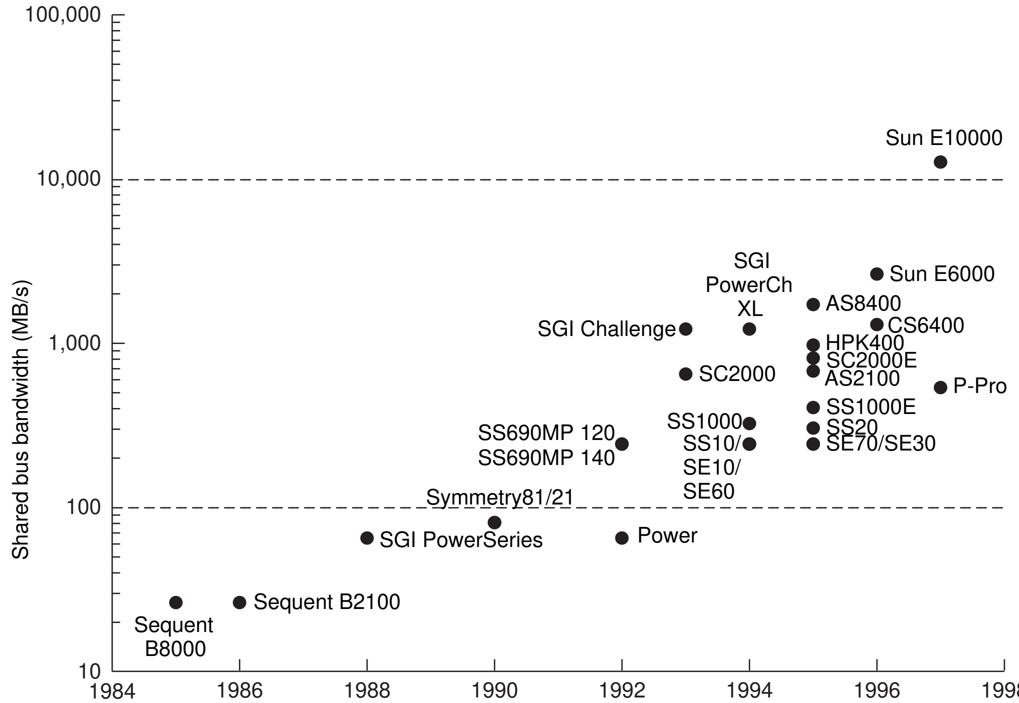

FIGURE 1.9 Bandwidth of the shared memory bus in commercial multiprocessors. After slow growth for several years, a new era of memory bus design began in 1991, which supported the use of substantial numbers of very fast microprocessors.

ally, a shift in the industry business model has been noted, where are being pushed by software vendors, especially database companie just by the hardware vendors. Combining these trends with the techno appears that the question is when, not if, multiple processors per prevalent.

1.1.4 Supercomputers

We have looked at the forces driving the development of parallel arc general market. A second, confluent set of forces comes from the qu absolute maximum performance, known as *supercomputing*. Although commercial and information processing applications are increasingly becoming i ers of the high end, scientific computing has historically been a ground for innovative architecture. In the mid-1960s, this inclu instruction processing and dynamic instruction scheduling, which place in microprocessors today. Starting in the mid-1970s, superc dominated by *vector processors*, which perform operations on sequences of da

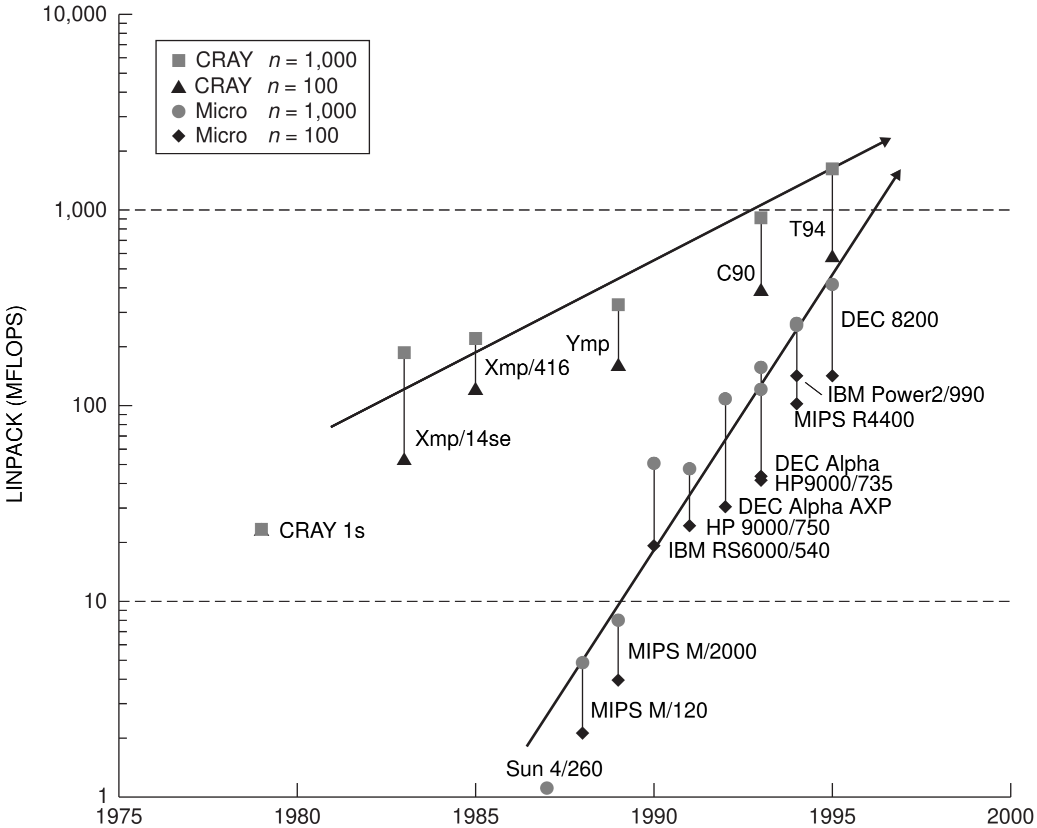

FIGURE 1.10 Uniprocessor performance of supercomputers and microprocessor-based systems on the LINPACK benchmark. Performance in MFLOPS for a single processor on solving dense linear equations is shown for the leading CRAY vector supercomputer and the fastest workstations on a 100×100 and $1{,}000 \times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

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

Dense linear algebra is an important component of scientific computing an specific emphasis of the LINPACK benchmark. Although this benchmark evaluates narrow aspect of system performance, it is one of the few measurements avail a very wide class of machines over a long period of time. Figure 1.10 sl LINPACK performance trend for one processor of the leading CRAY vector sup computers (August et al. 1989; Russel 1978) compared with that of the faste temporary microprocessor-based workstations and servers. For each system data points are provided. The lower one is the performance obtained 100×100 matrix and the higher one on a $1{,}000 \times 1{,}000$ matrix. Within the vector process approach, the single-processor performance improvement is dominatec improvements in cycle time and more substantial increases in the v bandwidth. In the microprocessor systems, we see the combined effect clock rate, using on-chip pipelined floating-point units, increasin size, increasing off-chip second-level cache size, and increasing u level parallelism. The gap in uniprocessor performance is rapidly cl

Multiprocessor architectures are adopted by both the vector proces processor designs, but the scale is quite different. The CRAY Xmp fir and then four processors, the Ymp eight, the C90 sixteen, and the The microprocessor-based supercomputers initially provided about 10 increasing to roughly 1,000 from 1990 on. These aggregations of proc as *massively parallel processors* (MPPs), have tracked the microprocessor adva with typically a lag of one to two years behind the leading micro workstation or personal computer. As shown in Figure 1.11, the lar slightly slower microprocessors has proved dominant for the LINPACK (Note the change of scale from MFLOPS in Figure 1.10 to GFLOPS here formance advantage of the MPP systems over traditional vector supe less substantial on more complete applications (Bailey et al. 1994) tive immaturity of the programming languages, compilers, and algorit the trend toward MPPs is still very pronounced. The importance of apparent enough in 1993 that CRAY Research announced its T3D, based Alpha microprocessor.

Recently, the LINPACK benchmark has been used to rank the fastes systems in the world. Figure 1.12 shows the number of multiprocesso tor processors (PVPs), MPPs, and bus-based shared memory multip (SMPs) appearing in the list of the top 500 systems. The latter two processor based, and the trend is clear.

1.1.5 Summary

In examining current trends from a variety of perspectives—economics architecture, and application demand—we see that parallel architect ingly attractive and increasingly central. The quest for performanc parallelism is being exploited at many different levels and at var computer design space. Instruction-level parallelism is exploited in performance processors. Essentially, all machines beyond the deskto cessors, including servers, mainframes, and supercomputers. The ver the performance curve is dominated by massively parallel processo large-scale parallelism in applications is broadening. The focus o multiprocessor level of parallelism. We study the design principles allel machines from the modest scale to the very large, so that we the spectrum of viable parallel architectures that can be built f components.

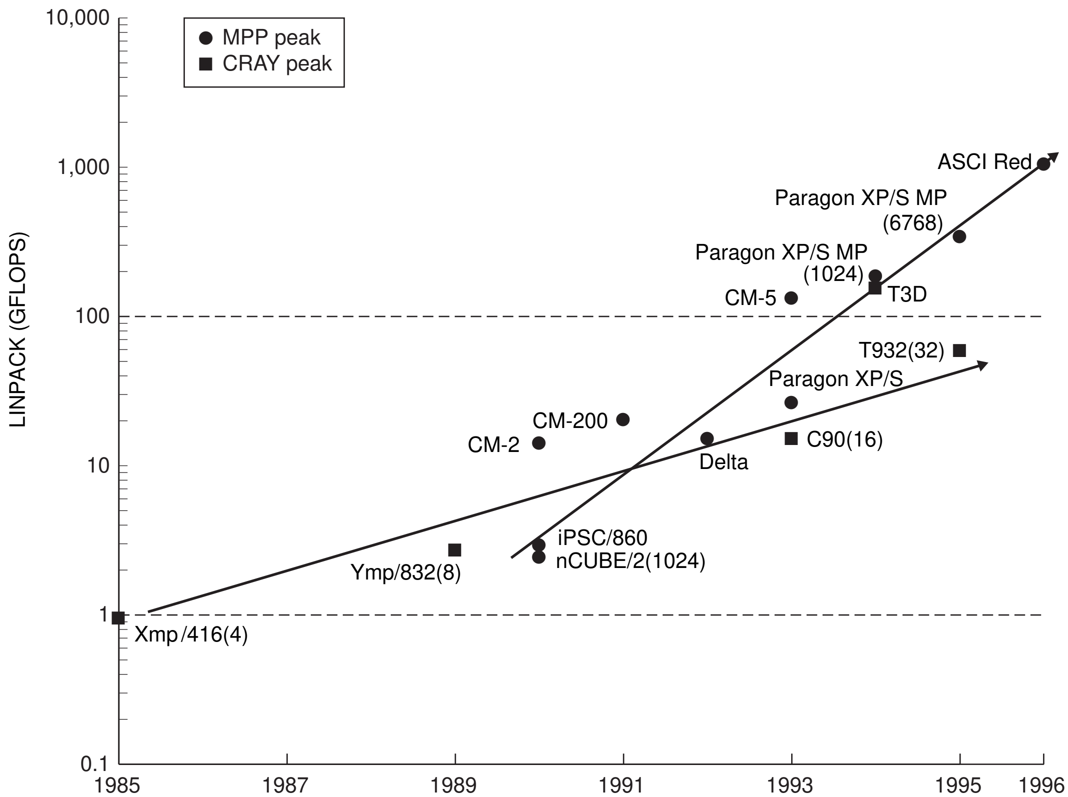

FIGURE 1.11 Performance of supercomputers and MPPs on the LINPACK peak performance benchmark. Peak performance in GFLOPS for solving dense linear equations is shown for the leading CRAY multiprocessor vector supercomputer and the fastest MPP systems. Note the change in scale from Figure 1.10 (MFLOPS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

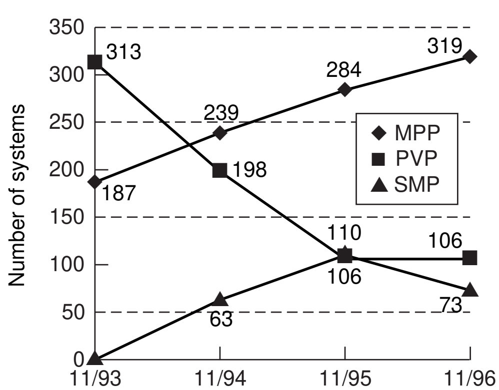

FIGURE 1.12 Types of systems used in the 500 fastest computer systems in the world. Parallel vector processors (PVPs) have given way to microprocessor-based massively parallel processors (MPPs) and bus-based symmetric shared memory multiprocessors (SMPs) at the high end of computing.

This discussion of the trends toward parallel computers has been the processor perspective, but you may arrive at the same conclusion ory system perspective. Consider briefly the design of a memory syste very large amount of data, that is, *large* data set. One of the few physical laws of computer architecture is that fast memories are sm ries are slow. This occurrence is due to many factors, including the decode time, the delays on increasingly long bit lines, the small dr dense storage cells, and the selector delays. The result is that m constructed as a hierarchy of increasingly larger and slower memorie large hierarchical memory is fast, as long as the references exhibit other trick we can play to cheat the laws of physics and obtain fas large data set is to use multiple processors and have the different independent smaller memories. Of course, physics is not easily fool cost when a processor accesses nonlocal data, *communication*, call when we need to orchestrate the actions of the many processors (i.e zation operations).

1.2 CONVERGENCE OF PARALLEL ARCHITECTURES

Historically, parallel machines have developed within several disti camps, and most texts on the subject are organized around a taxon designs. However, in looking at the evolution of parallel architectu the designs are strongly influenced by the same technological forc application requirements. It is not surprising therefore that a gr gence has occurred in the field. The goal of this section is to cons for understanding the entire spectrum of parallel computer archit build intuition as to the nature of the convergence. Along the way overview of the evolution of parallel machines, starting from the t and moving toward the point of convergence.

1.2.1 Communication Architecture

Given that a parallel computer is "a collection of processing eleme nicate and cooperate to solve large problems fast" (Almasi and Got may reasonably view parallel architecture as the extension of conven architecture to address issues of communication and cooperation amo elements. In essence, parallel architecture extends the usual concep architecture with a communication architecture. Computer architectur tinct facets. One is the definition of critical abstractions, espec software boundary and the user/system boundary. The architecture sp of operations at the boundary and the data types that these operat facet is the organizational structure that realizes these abstract performance in a cost-effective manner. A *communication architecture* has these two

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

The framework for understanding communication in a parallel machine is i trated in Figure 1.13. The top layer is the programming model, which is the tualization of the machine that the programmer uses in coding application programming model specifies how parts of the program running in parallel com nicate information to one another and what synchronization operations are a to coordinate their activities. Applications are written in a programming the simplest case, the model consists of multiprogramming a large number o pendent sequential programs; no communication or cooperation takes place a programming level. The more interesting cases include true parallel progra models, such as shared address space, message passing, and data parallel p ming. 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 know shared locations. Individual activities can be orchestrated by taking who is doing what task.
- *Message passing* is akin to telephone calls or letters, which convey informa from a specific sender to a specific receiver. There is a well-defined eve the information is sent or received, and these events are the basis fc trating individual activities. However, no shared location is accessibl
- *Data parallel* processing is a more regimented form of cooperation, where se eral agents perform an action on separate elements of a data set simulta and then exchange information globally before continuing en masse. The bal reorganization of data may be accomplished through accesses to sh addresses or messages since the programming model only defines the over effect of the parallel steps.

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

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

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

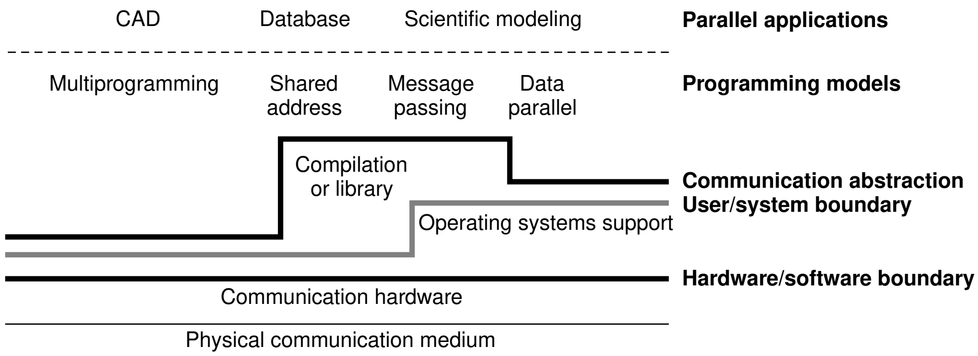

FIGURE 1.13 Layers of abstraction in parallel computer architecture. Critical layers of abstractions lie between the application program and the actual hardware. The application is written for a programming model, which dictates how pieces of the program share information and coordinate their activities. 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 pro
even in conventional instruction sets, some operations may be reali
nation of hardware and software, such as a load instruction that re
system intervention in the case of a page fault. The communication a
extends the computer organization with the hardware structures that
munication.

As with conventional computer architecture, a great deal of debat
over the years about what should be incorporated into each layer o
parallel architecture and how large the gap between the layers s
debate has been fueled by differing assumptions about the underlyi
and more qualitative assessments of "ease of programming." The hard
boundary in Figure 1.13 is depicted as flat to indicate that the av
primitives in different designs is more or less of uniform complexi
becoming more the case as the field matures. In most early designs
hardware organization was strongly oriented toward a particular
model; that is, the communication abstraction supported by the h
essentially identical to the programming model. This "high-level"
tecture approach resulted in tremendous diversity in the hardware
However, as the programming models have become better understood a
mentation techniques have matured, compilers and run-time libraries
provide an important bridge between the programming model and the
hardware. Simultaneously, the technological trends discussed in Sec
exerted a strong influence, regardless of the programming model. Tl

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

Sections 1.2.2–1.2.6 survey the most widely used programming models and corresponding styles of machine design in past and current parallel machine the historical orientation to a particular programming model, it was comm lump the programming model, the communication abstraction, and the machin organization together as "the architecture," for example, shared memory ar ture, message-passing architecture, and so on. This approach is less app today since a large commonality exists across parallel machines and sinc machines support several programming models. It is important to see how thi vergence has come about, so these sections begin from the traditional pers looking at machine designs associated with particular programming models explaining their intended roles and the technological opportunities that i their design. The goal of the survey is not to develop a taxonomy of machines per se but to identify a set of core concepts that form the basis ing design trade-offs across the entire spectrum of potential designs today future. It also demonstrates the influence that the dominant technological c established by microprocessor and DRAM technologies has had on parallel mac design, which makes a common treatment of the fundamental design issues nat or even imperative. Specifically, shared address, message-passing, data paral flow, and systolic approaches are presented. In each case, the abstraction e in the programming model is explained, and the reasons for the particular design, as well as the intended scale and application, are presented. The t cal motivations for the approach are also examined, as well as how they changed over time. These changes are reflected in the machine organization, determines what is fast and what is slow. The performance characteristics r to influence aspects of the programming model. The outcome of this brief surv clear organizational convergence, which is captured in a generic parallel m 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 implicitl result of conventional memory access instructions (i.e., loads and stores). has a long history, dating at least to precursors of mainframes[3] in the ear and today it has a role in almost every segment of the computer industry. memory multiprocessors serve to provide better throughput on multiprogramm workloads, as well as to support parallel programs. Thus, they are natural across a wide range of scale, from a few processors to perhaps hundreds. T

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

tion examines the communication architecture of shared memory machi key organizational issues for small-scale designs and large configura

The primary programming model for these machines is essentially t sharing on a single processor, except that real parallelism replac time. Formally, a *process* is a virtual address space and one or more threac trol. Processes can be configured so that portions of their address that is, are mapped to a common physical location, as suggested b (Multiple threads within a process, by definition, share portions space.) Cooperation and coordination among threads is accomplished and writing shared variables and pointers referring to shared addre *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 communication through shared addresses as well as special atomic c synchronization. Even completely independent processes typically sh portion of the address space, although this is only accessed by c code. Nonetheless, the shared address space model is utilized withi system to coordinate the execution of the processes.

Although shared memory can be used for communication among arbitra tions of processes, most parallel programs are quite structured in t tual address space. They typically have a common code image, privat the stack and other private data, and shared segments that are in t the virtual address space of each process or thread of the program. ture implies that the private variables in the program are present i that shared variables have the same address and meaning in each t straightforward parallelization strategies are employed. For exampl may perform a subset of the iterations of a common parallel loop or, processes may operate as a pool of workers obtaining work from a Chapter 2 discusses the structure of parallel programs more deeply. the basic evolution and development of this important architectural

The communication hardware for shared memory multiprocessors is extension of the memory system found in most computers. Essentially systems allow a processor and a set of I/O controllers to access memory modules through some kind of hardware interconnect, as ill Figure 1.15. The memory capacity is increased simply by adding memo Additional capacity may or may not increase the available memory depending on the specific system organization. I/O capacity is incre devices to I/O controllers or by inserting additional I/O controll possible ways to increase the processing capacity: wait for a fa become available or add more processors. On a time-sharing workloa processing capacity should increase the throughput of the system. W cessors, more processes can run at once and throughput is increas application is programmed to make use of multiple threads, more proc speed up the application. The hardware primitives are essentially the communication abstraction, and these operations are available i ming model.

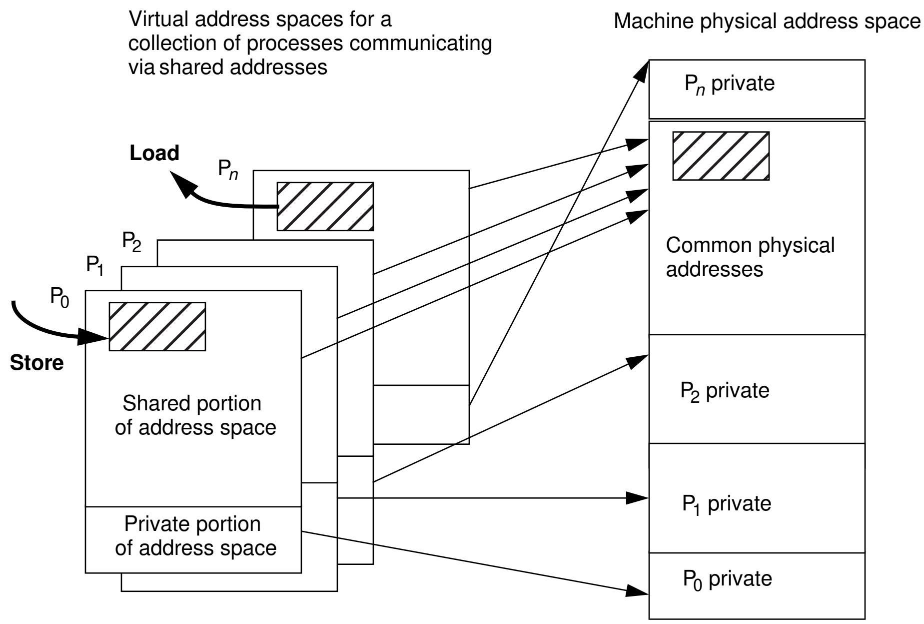

FIGURE 1.14 Typical memory model for shared memory parallel programs. Collections of processes have a common region of physical addresses mapped into their virtual address space, in addition to the private region, which typically contains the stack and private data.

Within the general framework of Figure 1.15, a great deal of evolution o
memory machines has taken place as the underlying technology has advanced.
early machines were "high-end" mainframe configurations (Lonergan and Kir
1961; Padegs 1981). On the technology side, memory in early mainframes was
compared to the processor, so it was necessary to interleave data across sev
ory banks to obtain adequate bandwidth for even a single processor; this re
interconnect between the processor and each of the banks. On the applicati
these systems were primarily designed for throughput on a large number of
Thus, to meet the I/O demands of a workload, several I/O channels and devic
attached. The I/O channels also required direct access to each of the memor
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
1.16a. Adding processors was primarily a matter of expanding the switch; th
ware structure to access a memory location from a port on the processor a
side of the switch was unchanged. The size and cost of the processor limit
early systems to a small number of processors, but as the hardware density
improved, larger systems could be contemplated. The cost of scaling the c
became the limiting factor, and in many cases it was replaced by a *multistage inter-connect*, suggested by Figure 1.16b, for which the cost increases more slowl

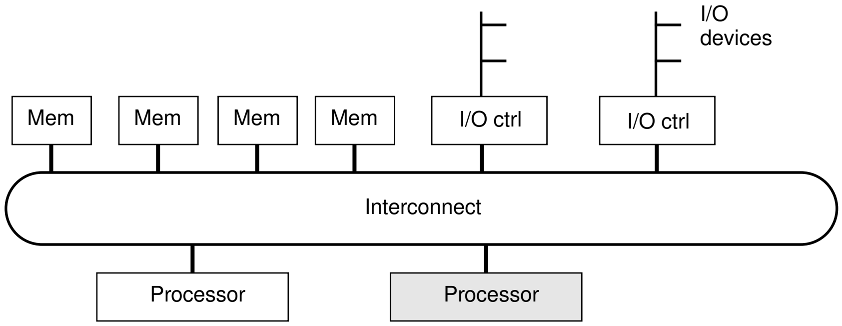

FIGURE 1.15 Extending a system into a shared memory multiprocessor by adding processor modules. Most systems consist of one or more memory modules accessible by a processor and I/O controllers through a hardware interconnect, typically a bus, crossbar, or multistage interconnect. Memory and I/O capacity 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).

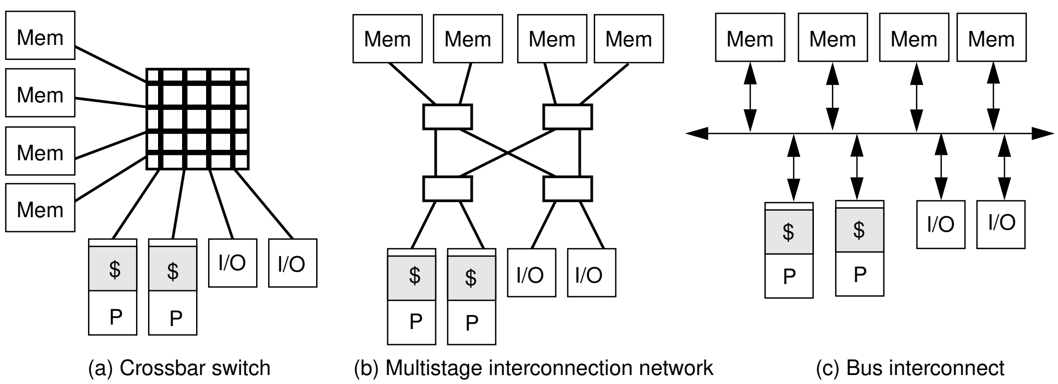

FIGURE 1.16 Typical shared memory multiprocessor interconnection schemes. The interconnection of multiple processors, with their local caches (indicated by $), and I/O controllers to multiple memory modules may be via crossbar, multistage interconnection network, or bus.

the number of ports. These savings come at the expense of increase
decreased bandwidth per port if all are used at once. The ability to
directly from each processor has several advantages: any processor c
cess or handle any I/O event, and data structures can be shared with
system.

The widespread use of shared memory multiprocessor designs came a
the 32-bit microprocessor revolution in the mid-1980s because the pr
floating-point, and memory management unit fit on a single board (Be
even two to a board. Most mid-range machines, including minicompute
workstations, and personal computers, are organized around a central
as illustrated in Figure 1.16c, and the bus could be adapted to s

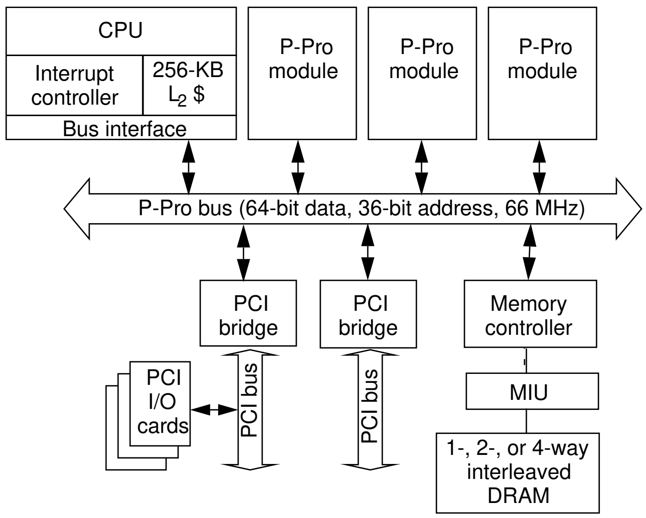

FIGURE 1.17(a) Physical and logical organization of the Intel Pentium Pro four-processor "quad pack." The Intel quad-processor Pentium Pro motherboard employed in many multiprocessor servers illustrates the major design elements of most small-scale shared memory multiprocessors. Its logical block diagram (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 acce physical address in the system. Like the switch-based designs, all memory l are equidistant to all processors, so all processors experience the same a 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 a well as multiprogramming. The typical organization of the bus-based symm multiprocessor is illustrated in more detail by Figure 1.17, which describe

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

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 h server organization that distributes the physical memory over the process ules, but retains symmetric access.

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

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

One natural approach to building scalable shared memory machines is to m tain the uniform memory access (or "dancehall") approach of Figure 1.15 an vide a scalable interconnect between the processors and the memories. memory access is translated into a message transaction over the network, mu might be translated to a bus transaction in the SMP designs. The primary di tage of this approach is that the round-trip network latency is experiencec memory access and a large bandwidth must be supplied to every processor.

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

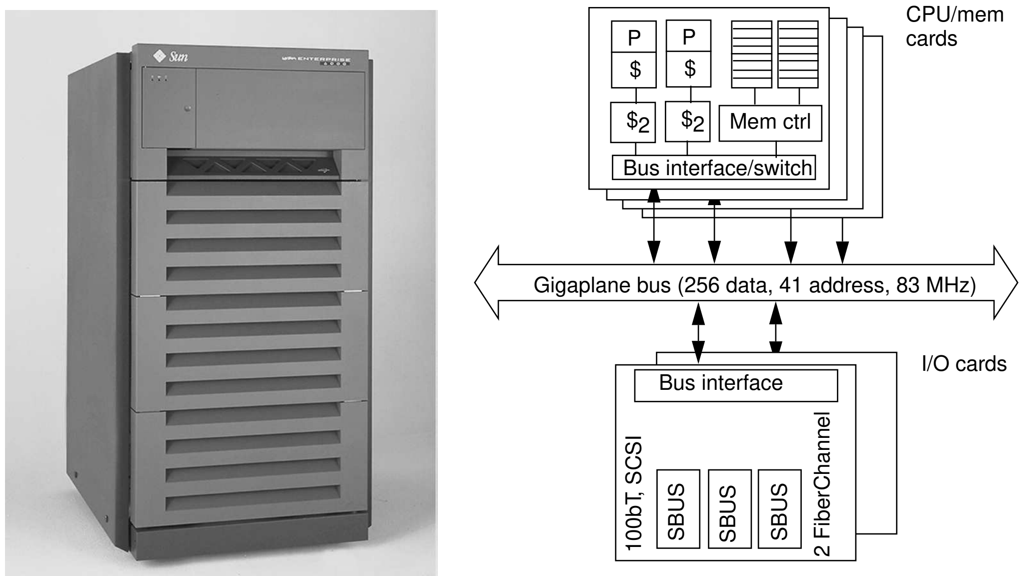

FIGURE 1.18 Physical and logical organization of the Sun Enterprise Server. A larger-scale design is illustrated by the Sun UltraSparc-based Enterprise multiprocessor server. The diagram shows its physical structure and logical organization. A wide (256-bit), highly pipelined memory bus delivers 2.5 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.

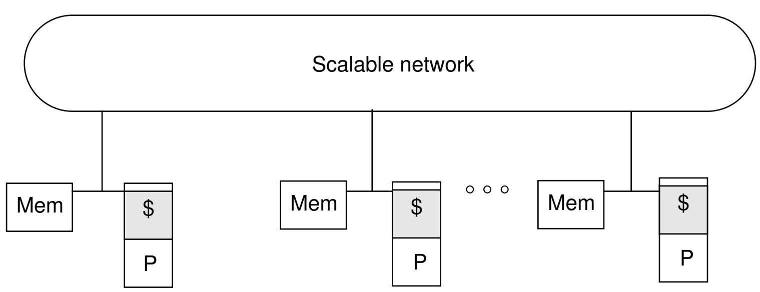

FIGURE 1.19 Nonuniform memory access (NUMA) scalable shared memory multiprocessor organization. Processor and memory modules are closely integrated such that access to local memory is faster than access to remote memories.

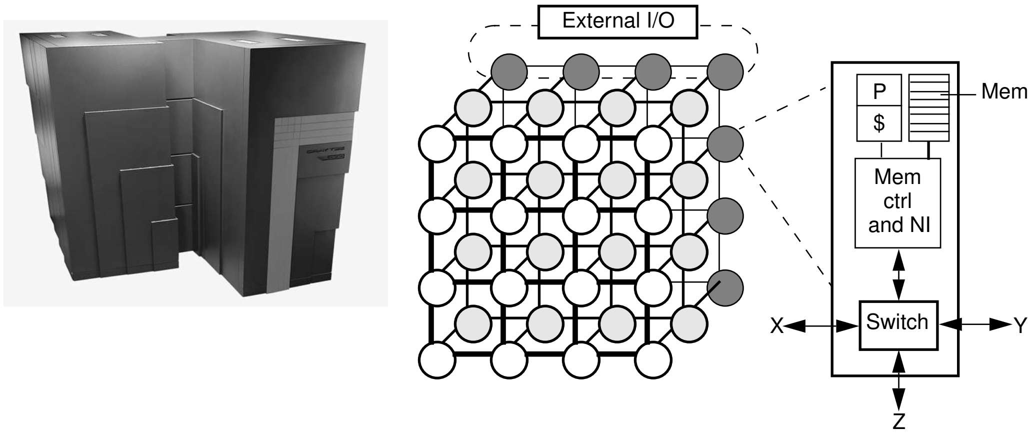

FIGURE 1.20 CRAY T3E scalable shared address space machine. The CRAY T3E is designed to scale up to a thousand processors supporting a global shared address space. Each node contains a DEC Alpha processor, local memory, a network interface integrated with the memory controller, and a network switch. 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 the accesses are to local data. The bandwidth demand placed on the network reduced. Although some conceptual simplicity arises from having all share equidistant from any processor, the NUMA approach has become far more preva at a large scale because of its inherent performance advantages and becaus nesses more of the mainstream processor memory system technology. One examp of this style of design is the CRAY T3E, illustrated in Figure 1.20. Thi reflects the viewpoint that, although all memory is accessible to every proc distribution of memory across processors is exposed to the programmer. Cach used only to hold data (and instructions) from local memory. It is the prog job to avoid frequent remote references. The SGI Origin is an example of a with a similar organizational structure, but it allows data from any memo replicated into any of the caches and provides hardware support to keep the consistent without relying on a bus connecting all the modules with a comm

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

To summarize, communication and cooperation in the shared address
gramming model consists of reads and writes to shared variables; t
are mapped directly to a communication abstraction consisting of l
instructions accessing a global, shared address space, which is sup
hardware through access to shared physical memory locations. The p
model and communication abstraction are very close to the actual h
processor can *name* every physical location in the machine; a process ca
data it shares with others within its virtual address space. Data i
as primitive types in the instruction set (bytes, words, etc.) or a
process performs memory operations on addresses in its virtual add
address translation process identifies a physical location, which
remote to the processor and may be shared with other processes. In
hardware accesses it directly, without user or operating system so
tion. The address translation realizes protection within the share
just as it does for uniprocessors, since a process can only access t
address space.

The effectiveness of the shared memory approach depends on the lat
on memory accesses as well as the bandwidth of data transfer that ca
Just as a memory storage hierarchy allows data that is bound to a
migrated toward the processor, expressing communication in terms o
address space allows shared data to be migrated toward the processor
However, migrating and replicating data across a general-purpose in
sents a unique set of challenges. We will see that to achieve sca
design, the entire solution, including the hardware interconnect m
for maintaining the consistent shared memory abstractions, must scal

1.2.3 Message Passing

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

In message passing, a substantial distance exists between the programming
and the actual hardware primitives, with user communication performed thr
operating system or library calls that perform many lower-level actions,
the actual communication operation. Thus, our discussion of message passing
with a look at the communication abstraction and then briefly surveys the ev
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 c
processor). *Receive* specifies a sending process and a local data buffer into which
transmitted data is to be placed. Together, the matching send and receive
data transfer from one process to another, as indicated in Figure 1.21.
message-passing systems, the send operation also allows a *tag* to be identifier or
attached to the message, and the receiving operation specifies a matching ru
as a specific tag from a specific processor, or any tag from any processor).
user program names local addresses and entries in an abstract process-tag s
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, dependi
whether the send completes when the receive has been executed, when the
buffer is available for reuse, or when the request has been accepted. Sim
receive can potentially wait until a matching send occurs or simply post th
Each of these variants has somewhat different semantics and different imple
tion requirements.

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

Early message-passing machines provided hardware primitives that were
close to the simple send/receive user-level communication abstraction, wit
additional restrictions. A node was connected to a fixed set of neighbors in
pattern by point-to-point links that behaved as simple FIFOs (Seitz 1985).
of design is illustrated in Figure 1.22 for a small 3D cube. Many early mach
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 con-
nected to neighbors on two or three dimensions. The network topology was
cially important in the early message-passing machines because only
neighboring processors could be named in a send or receive operation. The
transfer involved the sender writing to a link and the receiver reading fro
The FIFOs were small and so the sender would not be able to finish writing th

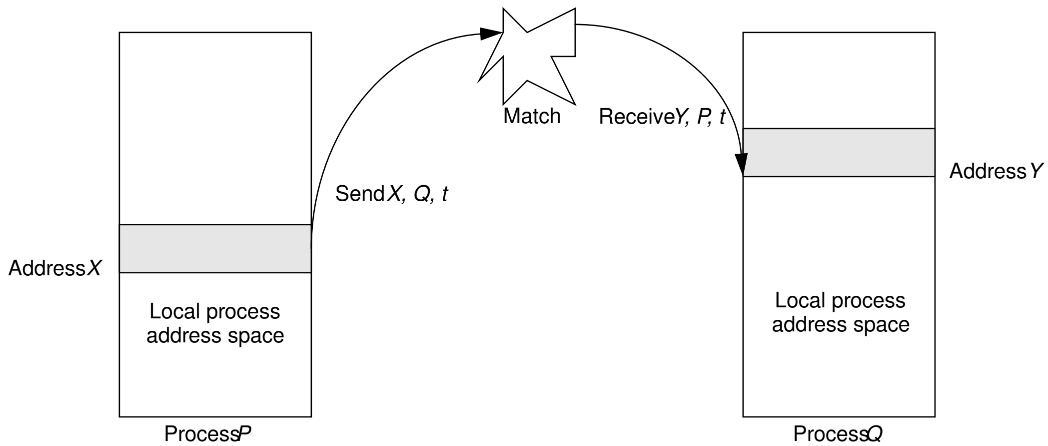

FIGURE 1.21 User-level send/receive message-passing abstraction. A data transfer from one local address space to another occurs when a send to a particular process is matched with a receive posted by that process.

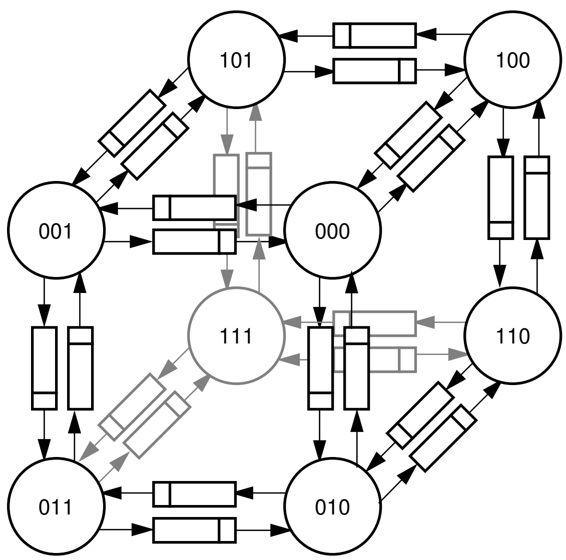

FIGURE 1.22 Typical structure of an early message-passing machine. Each node is connected to neighbors in three dimensions via FIFOs.

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

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

The direct FIFO design was soon replaced by more versatile and more ro designs that provided *direct memory access* (DMA) transfers on either end of the communication event. A DMA device is a special-purpose controller that tra data between memory and an I/O device without engaging the processor unti 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 pe receive) while the send completes. On the receiving end, the transfer is ac a DMA transfer by the message layer into a buffer and queued until the tar cess performs a matching receive, at which point the data is copying into t space of the receiving process.

The physical topology of the communication network so dominated the progra ming model of these early machines that parallel algorithms were often st terms of a specific interconnection topology, for example, a ring, a grid, c cube (Fox et al. 1988). However, to make the machines more generally usefu designers of the message layers provided support for communication between trary processors rather than only between physical neighbors. This was or supported by forwarding the data within the message layer along links in work. Soon this routing function was moved into the hardware (as discuss Chapter 10), so each node consisted of a processor with memory and a switc 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 t network, so an emphasis remained on interconnection topology. (See Exercis for a brief store-and-forward example.)

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

---

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

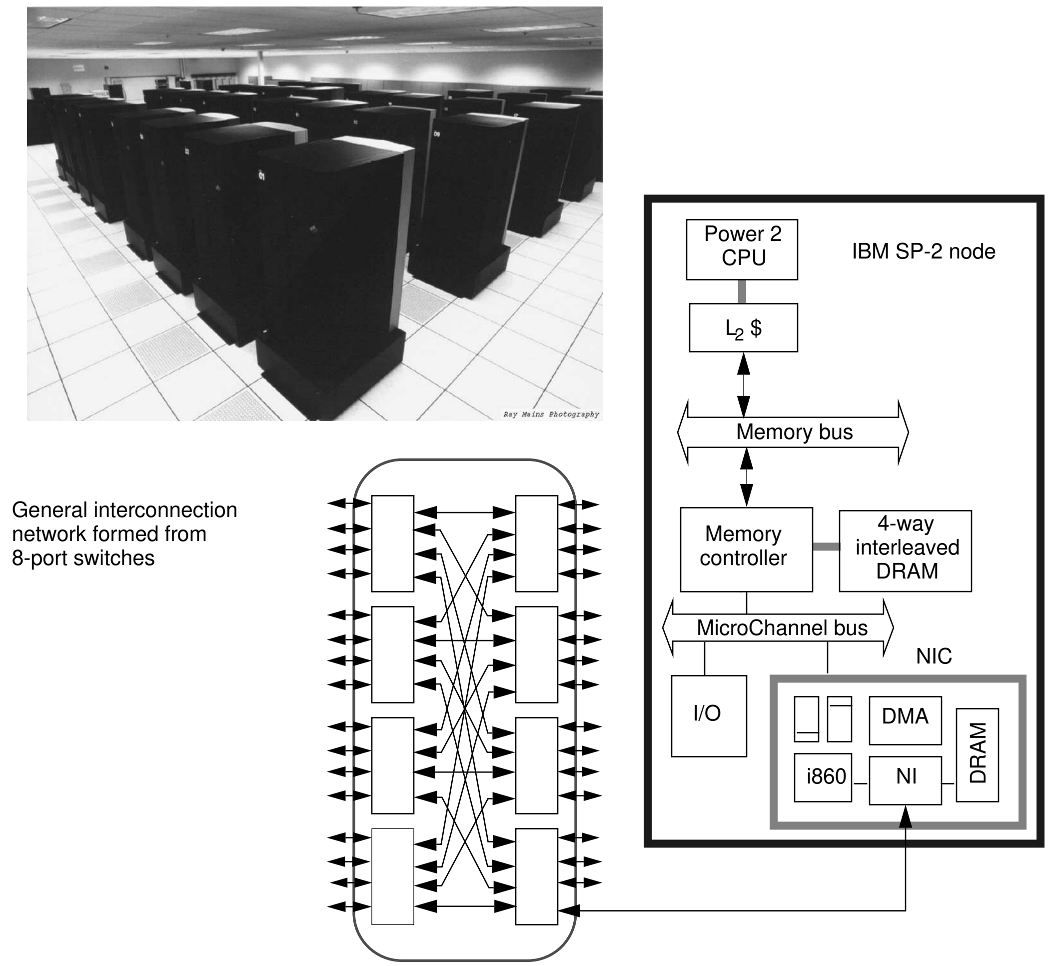

FIGURE 1.23 IBM SP-2 message-passing machine. The IBM SP-2 is a scalable parallel machine constructed essentially out of complete RS6000 workstations. Modest modifications are made to package the workstations into standing racks. A network interface card (NIC) is inserted at the MicroChannel I/O bus. 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 face more tightly to the processors in SMP nodes, where one of th dedicated to supporting message passing.

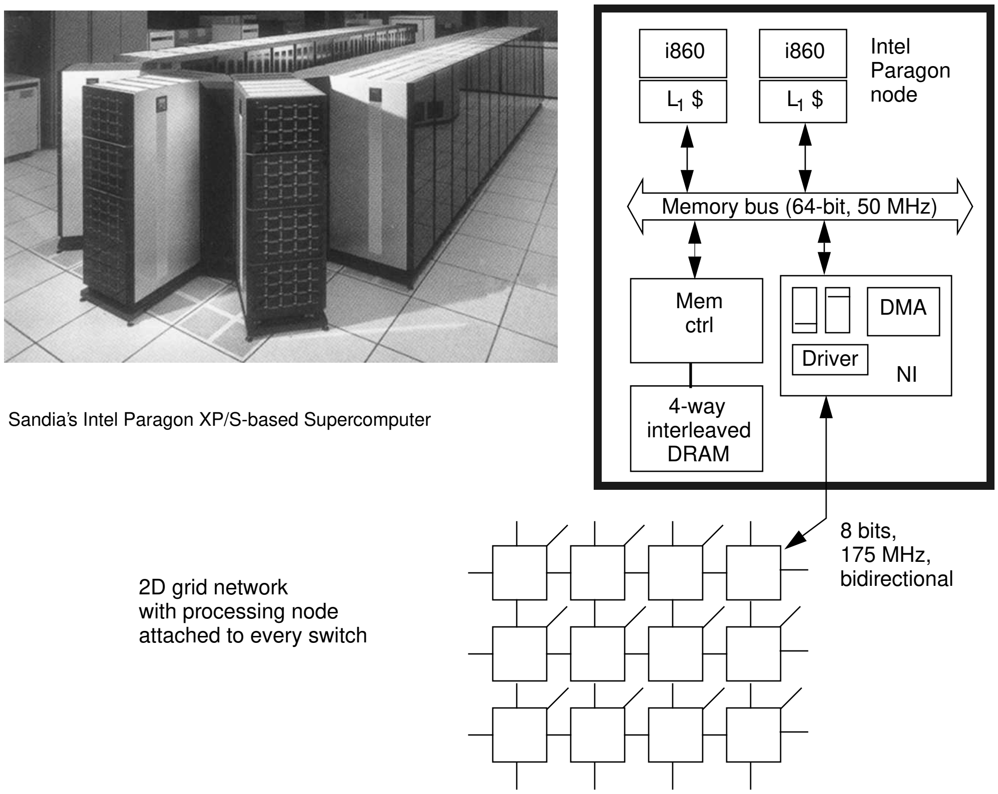

FIGURE 1.24 Intel Paragon. The Intel Paragon illustrates a much tighter packaging of nodes. Each card is an SMP with two or more i860 processors and a network interface chip connected to the cache-coherent memory bus. One of the processors is dedicated to servicing the network. In addition, the node 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 it memory and each of the processors, perhaps by number or by route. A user pr can only name private addresses and other processes; it can transfer data send/receive calls.

1.2.4 Convergence

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

- Traditional message-passing operations (send/receive) are suppo shared memory machines through shared buffer storage. Send invol data, or a pointer to data, into the buffer; receive involves from shared storage. Flags or locks are used to control access t to indicate events such as message arrival.
- On a message-passing machine, a user process may construct a gl space of sorts by carrying along pointers specifying the proces tual address in that process. Access to such a global address c in software through an explicit message transaction. Most mes libraries allow a process to accept a message for any process, can serve data requests from the others. A logical read is real request to the process containing the object and receiving a actual message transaction may be hidden from the user; it may b by compiler-generated code for access to a shared variable.
- A shared virtual address space can be established on a mess machine at the page level. A collection of processes has a re addresses but, for each process, only the pages that are local ble. Upon access to a missing (i.e., remote) page, a page fault operating system engages the remote node in a message transactio the page and map it into the user address space.

At the level of machine organization, substantial convergence ha well. Modern message-passing architectures appear essentially ident diagram level to the scalable NUMA design illustrated in Figure 1.1 memory case, the network interface was integrated with the cache memory controller in order for that device to observe cache misses a message transaction to access memory in a remote node. In the mes approach, the network interface is essentially an I/O device. Howev been to integrate this device more deeply into the memory system a transfer data directly from and to the user address space. Some desi transfers across the network, from memory on one machine to memory c machine, so the network interface is integrated fairly deeply with tem. Message passing is implemented on top of these remote memory ton, Crownie, and McLaren 1994). In some designs, a complete proces communication, sharing a cache-coherent memory bus with the main (Groscup 1992; Pierce and Regnier 1994). Viewing the convergence fr side, clearly all large-scale shared memory operations are ultimatel message transactions at some level.

In addition to the convergence of scalable message-passing and sh machines, switch-based local area networks, including fast Etherne Channel, and several proprietary designs (Boden et al. 1995; Gill emerged, providing scalable interconnects that are approaching what allel machines offer. These new networks are being used to connect machines (which may be shared memory multiprocessors in their own *clusters*, which may operate as a parallel machine on individual large problems many individual machines on a multiprogramming load. Essentially all SMP ver provide some form of network clustering to obtain better reliability.

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

1.2.5 Data Parallel Processing

A third important class of parallel machines has been variously called p arrays, single-instruction-multiple-data machines, and data parallel arch. The changing names reflect a gradual separation of the user-level abstracti 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. general paradigm have been many novel designs, exploiting various technolo opportunities, and considerable evolution as microprocessor technology has l such a dominant force.

An influential paper in the early 1970s (Flynn 1972) developed a taxonom 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 e they operate on: conventional sequential computers being *single-instruction-single-data* (SISD) and parallel machines built from multiple conventional processors *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 fet much in time and hardware as performing the actual instruction. The idea wa all the instruction sequencing could be consolidated in the control proces data processors included only the ALU, memory, and a simple connection to n neighbors.

In the SIMD machines, the data parallel programming model was rendere directly in the physical hardware (Ball et al. 1962; Bouknight et al. 197 1972; Reddaway 1973; Slotnick, Borck, and McReynolds 1962; Slotnick 1967; and Cornell 1978). Typically, a control processor broadcasts each instructi array of data processing elements (PEs), which are connected to form a regu

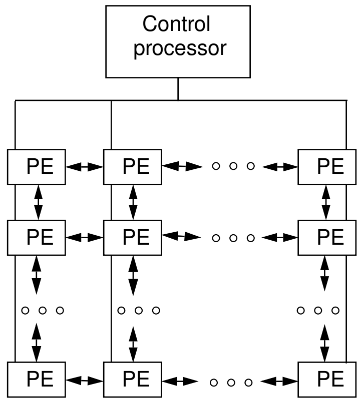

FIGURE 1.25 Typical organization of a data parallel (SIMD) machine. Individual processing elements (PEs) operate in lockstep under the direction of a single control processor. Traditionally, SIMD machines have provided a limited, regular interconnect among the PEs, although this was generalized in later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

as suggested by Figure 1.25. It was observed that many important sc
tations involved uniform calculation on every element of an array
involving neighboring elements in the row or column. Thus, the par
data was distributed over the memories of the data processors, and
retained in the control processor's memory. The control processor
data processors to each perform an operation on local data element
form a communication operation. For example, to average each elemen
with its four neighbors, a copy of the matrix would be shifted acros
of the four directions and a local accumulation performed in each P
ically included a condition flag, allowing some to abstain from an op
designs, the local address could be specified with an indirect ad
allowing all processors to do the same operation but with diffe
addresses.

The development of arrays of processors was almost completely ec
mid-1970s with the development of vector processors. In these mach
processor is integrated with a collection of function units that op
data out of one memory in a pipelined fashion. The ability to operat
where in memory eliminated the need to map application data struc
rigid interconnection structure and greatly simplified the problem
aligned so that local operations could be performed. The first vect
CDC Star-100, provided vector operations in its instruction set th
source vectors from memory and produced a result vector in memory.
only operated at full speed if the vectors were contiguous, and hen
of the execution time was spent simply transposing matrices. A dr
occurred in 1976 with the introduction of the CRAY-1, which extende
of a load-store architecture employed in the CDC 6600 and CDC 7600
covered in modern RISC machines) to apply to vectors. Vectors in m
fixed stride, were transferred to or from contiguous vector registe
and store instructions. Arithmetic was performed on the vector regi
a very fast scalar processor (operating at the unprecedented rate o

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

The SIMD data parallel machine experienced a renaissance in the mid-1980 VLSI advances made simple 32-bit processors just barely practical (Batche 1980; Hillis 1985; Nickolls 1990; Tucker and Robertson 1988). The unique tw the data parallel regime was to place 32 very simple 1-bit processing ele each chip, along with serial connections to neighboring processors, while dating the instruction sequencing capability in the control processor. In systems with several thousand bit-serial processing elements could be const reasonable cost. In addition, it was recognized that the utility of such a be increased dramatically with the provision of a general interconnect all arbitrary communication pattern to take place in a single, rather long ste tion to the regular grid neighbor connections (Hillis 1985; Hillis and St Nickolls 1990). The sequencing mechanism that expanded conventional integer floating-point operations into a sequence of bit-serial operations also p means of "virtualizing" the processing elements, so that a few thousand pr elements could give the illusion of operating in parallel on millions of da with one virtual PE per data element.

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

Thus, while the user-level abstraction of parallel operations on large re structures continued to offer an attractive solution to an important clas lems, the machine organization employed with data parallel programming mod evolved toward a more generic parallel architecture of multiple cooperatin processors, much like scalable shared memory and message-passing machin although several designs maintain specialized network support for global sy zation. One such example of network support is a *barrier*, which causes each pro- cess to wait at a particular point in the program until all other proce reached that point (Horiw et al. 1993; Leiserson et al. 1996; Kumar 1992; and Schwarzmeier 1993; Koeninger, Furtney, and Walker 1994). Indeed, the S approach evolved into the SPMD (single-program-multiple-data) approach,

which all processors execute copies of the same program, and has
converged with the more structured forms of shared memory and messa
programming.

Data parallel programming languages are usually implemented by v
local address spaces of a collection of processes, one per process
explicit global address space. Data structures are laid out across
space with a simple mapping from indexes to processor and local offs
tation is organized as a sequence of "bulk synchronous" phases of e
putation or global communication, separated by a global barrier (
Because all processors perform communication together and share a g
what is going on, either a shared address space or message passing c
For example, if a phase involved every processor doing a write to a
processor "to the left," it could be realized by each doing a send
receive "from the right" into the destination address. Similarly,
doing a read can be realized by every processor sending the address
processor sending back the data. In fact, the code that is produced
modern data parallel languages is essentially the same as for the s
parallel programs that are most common in shared memory and messag
programming models. The convergence in machine structure has been ac
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
received considerable investigation by academia and industry, but en
mercial success than the three classes just discussed and therefore
use as a vehicle for parallel programming. Two approaches that were
complete programming systems were dataflow architectures and systol
tures. Both represent important conceptual developments of continuin
field evolves.

Dataflow Architecture

Dataflow models of computation sought to make the essential aspects
computation explicit at the machine level, without imposing artific
that would limit the available parallelism in the program. The idea
gram is represented by a graph of essential data dependences, as
Figure 1.26, rather than as a fixed collection of explicitly sequenc
trol. An instruction may execute whenever its data operands are avai
may be spread arbitrarily over a collection of processors. Each n
operation to perform and the address of each of the nodes that need
original form, a processor in a dataflow machine operates as a simpl
line. 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 mat

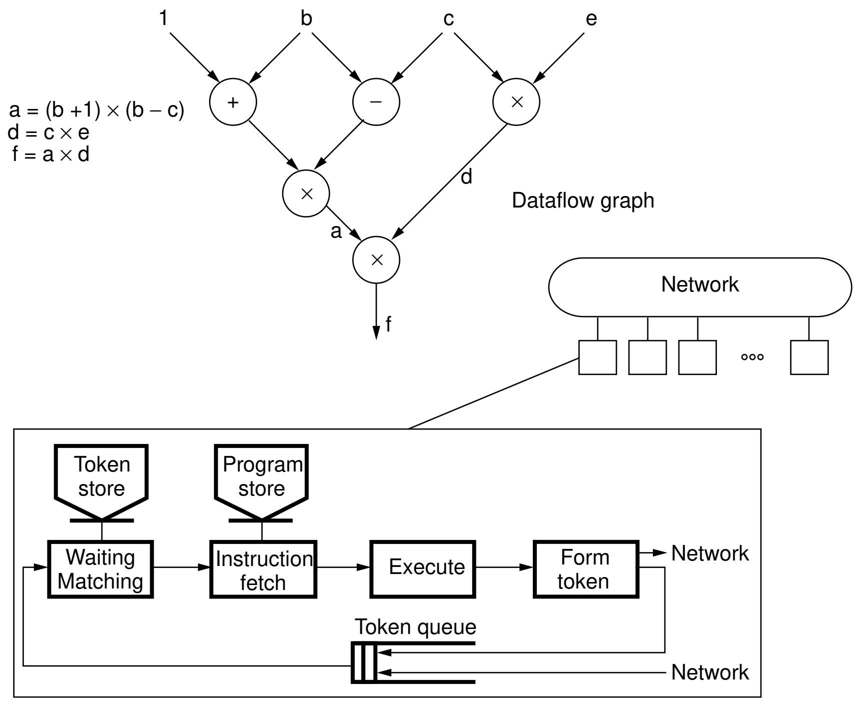

FIGURE 1.26 Dataflow graph and basic execution pipeline. A node in the graph fires when operands are present on its input. It produces results on its outputs that are delivered to adjacent nodes in the graph. The execution pipeline implements this firing rule by detecting when matching data tokens are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

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

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

The key characteristics of dataflow architectures are the ability to nam tions performed anywhere in the machine, the support for synchronization of pendent operations, and dynamic scheduling at the machine level. As the da machine designs matured into real systems programmed in high-level para languages, a more conventional structure emerged. Typically, paralle ated in the program as a result of parallel function calls and para attractive to allocate these larger chunks of work to processors. T of designs organized essentially like the NUMA design of Figure 1.19 entiating features being direct support for a large, dynamic set of and the integration of communication with thread generation. The closely integrated with the processor; in many designs, the "curr available in special registers, and hardware support is available f thread identified in the message. In addition, many designs provide on memory locations in order to provide fine-grained synchronization nization on an element-by-element basis) rather than using locks t accesses to an entire data structure. In particular, each message chunk of computation that could make use of local registers and memo

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

Systolic Architectures

Another novel approach was *systolic architectures*, which sought to replace a singl sequential processor by a regular array of simple processing elemen fully orchestrating the flow of data between PEs, obtain very high t modest memory bandwidth requirements. These designs differ from co pipelined function units in that the array structure can be nonline nal), the pathways between PEs may be multidirectional, and each P small amount of local instruction and data memory. They differ from each PE might do a different operation.

The early proposals were driven by the opportunity offered by VL inexpensive special-purpose chips. A given algorithm could be repre as a collection of specialized computational units connected in a efficient pattern. Data would move through the system at regular "b determined by local state. Figure 1.27 illustrates a design for co tions using a simple linear array. At each beat the input data advan multiplied by a local weight, and is accumulated into the output se

$y(i) = w1 \times x(i) + w2 \times x(i+1) + w3 \times x(i+2) + w4 \times x(i+3)$

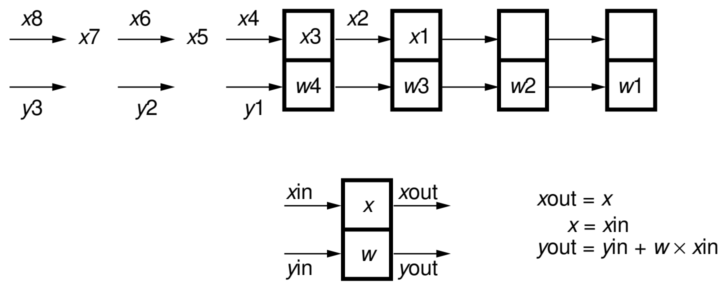

FIGURE 1.27 Systolic array computation of an inner product Each box represents a computational unit performing a specific function. Every time the clock beats, all units accept inputs, compute results, and generate outputs. Data moves through the systolic array with each beat.

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

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

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

1.2.7 A Generic Parallel Architecture

In examining the evolution of the major approaches to parallel architecture, clear convergence for scalable machines toward a generic parallel machine o tion, illustrated in Figure 1.28. The machine comprises a collection of e complete computers, each with one or more processors and memory, connecte through a scalable communication n *communication assist* a—controller

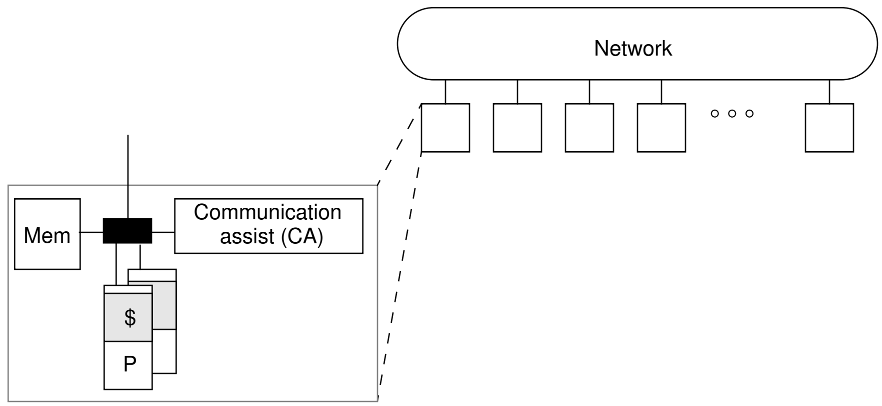

FIGURE 1.28 Generic scalable multiprocessor organization. A collection of essentially complete computers, including one or more processors and memory, communicating through a general-purpose, high-performance, scalable interconnect. Typically, each node contains a controller that assists in communication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

or auxiliary processing unit that assists in generating outgoing mes incoming messages. While the consolidation within the field may see the design space, in fact, great diversity and debate remains, cent tionality should be provided within the assist and how it interface memory system, and network. Recognizing that these are specific diff a largely similar organization helps us to understand and evaluat organizational trade-offs.

Not surprisingly, different programming models place different re the design of the communication assist and influence which operation and should be optimized. In the shared memory case, the assist is t with the memory system in order to capture the memory events that interaction with other nodes. The assist must also accept message memory operations and state transitions on behalf of other nodes. I passing case, communication is initiated by explicit actions, eithe user level, so it is not required that memory system events be obs need exists to initiate the messages quickly and to respond to inc The response may require that a tag match be performed, that buffe that data transfer commence, or that an event be posted. The data p tolic approaches place an emphasis on fast global synchronization, supported directly in the network or in the assist. Dataflow places fast dynamic scheduling of computation based on an incoming messa algorithms present the opportunity to exploit global patterns in I Even with these differences, it is important to observe that all o share common aspects; they need to initiate network transactions as cific processor events, and they need to perform simple operations node to carry out the desired event.

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

1.3 FUNDAMENTAL DESIGN ISSUES

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

The guiding principles presented in this book for understanding modern pa architecture are indicated by the layers of abstraction shown in Figure 1.1 mentally, we must understand the operations that are provided at the user-le munication abstraction, how various programming models are mapped to the primitives, and how these primitives are mapped to the actual hardware. Ex emphasis on the high-level programming model without attention to how it c mapped to the machine would detract from understanding the fundamental arc tectural issues, as would excessive emphasis on the specific hardware mechan each particular machine.

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

1.3.1 Communication Abstraction

The communication abstraction forms the key interface between the p
model and the system implementation. It plays a role very much like
set in conventional sequential computer architecture. Viewed from th
it must have a precise, well-defined meaning so that the same prograr
rectly on many implementations. In addition, the operations provide
must be simple, composable entities with clear costs, so that the
optimized for performance. Viewed from the hardware side, it must al
defined meaning so that the machine designer can determine where pe
optimizations can be performed without violating the software assum
the abstraction needs to be precise, the machine designer would l
overly specific, so it does not prohibit useful techniques for perfc
ment or frustrate efforts to exploit properties of newer technologie

The communication abstraction is, in effect, a contract between th
the software allowing each the flexibility to improve what it does wh
rectly together. To understand the "terms" of this contract, we ne
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 opera
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 fo
cessor. A thread can name the locations in its virtual address spa
machine registers. In some systems, the address space is broken u
code, stack, and heap segments whereas in others it is flat. Simila
gramming languages provide access to the address space in different
ple, some allow pointers and dynamic storage allocation, others do n
these variations, the instruction set provides the operations that c
the named locations. For example, in RISC machines the thread can l
or store data to memory but can perform arithmetic and comparisons o
registers. Older instruction sets support arithmetic on either. Cc
mask these differences at the hardware/software boundary, so the user
model is one of performing operations on variables that hold data.
translates each virtual address to a physical address on every opera

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

because the compiler performs optimizations when translating the program t instruction set and the hardware performs optimizations when executing the i tions. Both make sure the program cannot tell that the order has been chang 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 c uses the proper value, but they may avoid actually writing and reading the and from memory or may defer the write until later. Collections of reads intervening writes may be completely reordered and, generally, writes to c addresses can be reordered as long as dependences from intervening read preserved. This reordering occurs at the compilation level, for example, w compiler allocates variables to registers, manipulates expressions to impr lining, or transforms loops to reduce overhead and improve the data access It occurs at the machine level when instruction execution is pipelined, instructions are issued per cycle, or write buffers are used to hide memor We depend on these optimizations for performance. They work because for the gram to observe the effect of a write, it must read the variable; this crea dence, which is preserved. Thus, the illusion of program order is preserv actually executing the program in the weaker dependence order.[6] We operate in a world where essentially all programming languages embody a programming mod of sequential order of operations on variables in a virtual address space, tem enforces a weaker order wherever it can do so without changing the res the program.

Now let's return to parallel programming models. The informal discussion in this chapter indicated the distinct positions adopted on naming, opera and ordering. Naming and operation set are what typically characterize the however, ordering is of key importance. A parallel program must coordinat activity of its threads to ensure that the dependences within the prog 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 performance. We can focus on shared address and message-passing programmi models since they are the most widely used; other models, such as data para usually implemented in terms of one of them.

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

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

---

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

cessor operations are provided on the private address space, in prc
additional operations, send and receive, operate on the local addre
global process space. Send transfers data from the local address sp
Receive accepts data into the local address space from a process. E
pair is a specific point-to-point synchronization operation. Many m
languages offer global, or *collective*, communication operations as well, such
broadcast.

Naming

The position adopted on naming in the programming model is presentec
grammer through the programming language or programming environmer
what the logic of the program is based upon. However, the issue of n
at each level of the communication architecture. Certainly one possi
have the operations in the programming model be one to one with the
the communication abstraction at the user/system boundary and to h
one to one with the hardware primitives. However, it is also possil
piler and libraries to provide a level of translation between the p
and the communication abstraction, or for the operating system to ir
dle some of the operations at the user/system boundary. These alterr
architect to consider implementing the common, simple operations dir
ware and supporting the more complex operations partly or wholly in

Let us consider the ramifications of naming at the layers using tl
programming models: shared address and message passing. First, in a
model, accesses to shared variables in the program are usually mapp
piler to load and store instructions on shared virtual addresses,
any other variable. This is not the only option, however. The compi
ate special code sequences for accesses to shared variables. A mac
global physical address space if any processor is able to generate a physical
any location in the machine and access the location in a single memc
is straightforward to realize a shared virtual address space on a ma
global physical address space: establish the virtual-to-physical
shared virtual addresses map to the same physical location (i.e., t
the same entries in their page tables). However, the existence of tl
tion allows for other approaches. A machine supports *independent local physical address spaces* if each processor can only access a distinct set of loca
such a machine, a shared virtual address space can be provided by m
addresses that are local to a process to the corresponding physical
local addresses are left unmapped so upon access to a nonlocal share
fault will occur, allowing the operating system to intervene and a
shared data. Although this approach can provide the same naming, op
ordering to the program, it clearly has different hardware requi
hardware/software boundary. The architect’s job is to resolve these
across layers of the system implementation so that the result is e
effective for the target application workload on available technolog

Second, message-passing operations could be realized directly in hardwar the matching and buffering aspects of the send/receive operations are bett to software implementation. More basic data transport primitives are well s in hardware. Thus, in essentially all parallel machines, the message-pass gramming model is realized via a software layer that is built upon a simple nication abstraction. At the user/system boundary, one approach is to h message operations go through the operating system as if they were I/O ope: However, the frequency of message operations is much greater than I/O opera so it makes sense to use the operating system support to set up resources,] and so on and allow the frequent, simple data transfer operations to be s directly in hardware. On the other hand, we might consider adopting a shar tual address space as the lower-level communication abstraction, in which c and receive operations involve writing and reading shared buffers and post appropriate synchronization events.

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

Operations

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

Ordering

The properties of the specified order among operations have a profound e throughout the layers of parallel architecture. Notice, for example, that tl passing model places no assumption on the ordering of operations by distin cesses except the explicit program order associated with the send/receive o

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

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

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

- *Mutual exclusion* ensures that certain operations on certain data are
by only one thread or process at a time. We can imagine a room
entered to perform such an operation, and only one process can b
at a time. This is accomplished by locking the door upon entry
it on exit. If several processes arrive at the door together, c
and the others will wait until it leaves. The order in which t
allowed to enter does not matter and may vary from one executio
gram to the next; what matters is that they do so one at a time
sion operations to serialize the execution of processes.
- *Events* are used to inform other processes that some point of ex
been reached so that they can proceed knowing that certain deper
been satisfied. These operations are like passing a baton from
the next in a relay race or the starter firing a gun to indicate
If one process writes a value that another is supposed to read
chronization operation must take place to indicate that the valu
read. Events may be *point-to-point*, involving a pair of processes, or the
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 architectur nication and data replication. Communication and replication are inherently Consider first a message-passing operation. The effect of the send/receive copy data that is in the sender's address space into a region of the recei space. This transfer is essential for the receiver to access the data. If duced by the sender, it represents a *true communication* of information from one process to the other. If the data just happens to be stored at the sender, perhaps was the initial configuration of the data or because the data set was simply to fit on any one node, then this transfer merely makes a replica of the dat is used. In this case, the processes are not actually communicating informa one to another via the data transfer. If the data were replicated or posit erly over the processes to begin with, there would be no need to communicat message. More importantly, if the receiver uses the data over and over aga reuse its replica without additional data transfers. The sender can modify of addresses that was previously communicated with no effect on the pre receiver. If the effect of these later updates is to be communicated, an transfer must occur.

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

Interprocess communication and data transfer within the storage hierarchy melded together in a shared physical address space. Cache misses cause a da fer across the machine interconnect whenever the physical backing storage address is remote to the node accessing the address, whether the address i or shared and whether the transfer is a result of true communication or ju access. The natural tendency of the machine is to replicate data into the the processors that access the data. If the data is reused while it is in data transfers occur; this is a major advantage. However, when a write to s occurs, something must be done to ensure that later reads by other process the new data rather than the old data that was replicated into their caches involve more than a simple data transfer.

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

processor can observe. If it is updated, the other hidden replica
actual memory location, must eventually be updated. Copying data bi
of addresses to the same set of values. Generally, this will cause c
the copy has been made, the two sets of bindings are completely
(unlike the implicit replication that occurs within the storage hier
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 ma
transfer within the machine, either on the write or the read, or the
occur for other reasons. Communication may involve establishing a n
not doing so, depending on the particular communication abstraction.

In general, replication avoids unnecessary communication; that i
data to a consumer that was not produced since the data was previo
The ability to perform replication automatically at a given level c
tion architecture depends very strongly on the naming and ordering
the layer. Moreover, replication is not a panacea—it too requires da
disadvantageous to replicate data that is not going to be used. We w
cation plays an important role throughout parallel computer architec

1.3.4 Performance

In defining the set of operations for communication and cooperation,
and the addressing modes, the communication abstraction specifies
objects are named, what ordering properties are preserved, and how s
is performed. However, the performance characteristics of the avai
determine how they are actually used. Programmers and compiler write
costly operations where possible. In evaluating architectural trade
between feasible alternatives ultimately rests upon the performanc
Thus, to complete an introduction to the fundamental issues of par
architecture, we need a framework for understanding performance at m
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
world where processors do only one thing at a time, these metric
related—the bandwidth (operations per second) is the reciprocal of t
onds per operation), and the cost is simply the latency times the n
tions performed. However, modern computer systems do many different
at once, and the relationship between these performance metrics i
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 communica the operations that we are concerned with most often are data transfers. Th mance of these operations can be understood as a generalization of our bas line example.

Data Transfer Time

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

$$TransferTime(n) = T_0 + \frac{n}{B} \tag{1.3}$$

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), a constant term, T_0, is the start-up cost. This is a very convenient model, and it to describe a diverse collection of operations, including messages, memory bus transactions, and vector operations. For message passing, the start-up be thought of as the time for the first bit to get to the destination. Fo operations, it is essentially the access time. For bus transactions, it re arbitration and command phases. For any sort of pipelined operation, inc pipelined instruction processing or vector operations, it is the time t pipeline.

Using this simple model, it is clear that the bandwidth of a data transfe depends on the transfer size. As the transfer size increases, it approaches totic 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 whic peak bandwidth is obtained, the *half-power point,* is given by

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

Unfortunately, this linear model does not give any indication of whe
operation can be initiated, nor does it indicate whether other use
performed during the transfer. These other factors depend on how t
performed.

Overhead and Occupancy

The data transfer in which we are most interested is the one that c
network in parallel machines. It is initiated by the processor throu
cation assist. The essential components of this operation can be des
lowing simple model:

$$\mathit{CommunicationTime}(n) = \mathit{Overhead} + \mathit{Occupancy} + \mathit{Network\ Del} \quad (1.5)$$

The *overhead* is the time the processor spends initiating the transfe
a fixed cost, if the processor simply has to tell the communication a
may be linear in n, if the processor has to copy the data into the assist.
is that this is time the processor is busy with the communication ev
other useful work or initiate other communication during this time.
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 s
nent on the communication path. For example, each link that is trave
work will be occupied for n/B, where B is the bandwidth of the link. The d
will occupy other resources, including buffers, switches, and the
assist. Often the communication assist is the bottleneck that dete
pancy. The occupancy limits how frequently communication operations
ated. The next data transfer will have to wait until the critical r
occupied before it can use that same resource. If there is bufferin
cessor and the bottleneck, the processor may be able to issue a burs
frequency greater than $1/\mathit{Occupancy}$; however, once this buffer is full, the
must slow to the rate set by the occupancy. A new transfer can sta
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
other factors, such as the time to get through the communication as
processor's viewpoint, the specific hardware components contributin
delay are indistinguishable. What affects the processor is how lo
before it can use the result of a communication event, how much of
use for other activities, and how frequently it can communicate dat
task of designing the network and its interfaces is very concerned
components and their contribution to the aspects of performance tha
observes.

In the simple case where the processor issues a request and waits
the breakdown of the communication time into its three components i

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

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

Equation 1.5 is a very general model. It can be used to describe data tr many places in modern, highly pipelined computer systems. As one example, sider the time to move a block between cache and memory on a miss. The ca controller spends a period of time inspecting the tag to determine that it and then starting the transfer; this is the overhead. The occupancy is the divided by the bus bandwidth, unless there is some slower component in the s The delay includes the normal time to arbitrate and gain access to the bus time spent delivering data into the memory. Additional time spent waiting access to the bus or waiting for the memory bank cycle to complete is due to tion. A second obvious example is the time to transfer a message from one p to another.

Communication Cost

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

$$CommunicationCost = Frequency \times (CommunicationTime - Overlap) \quad (1.6)$$

The *frequency of communication*, defined as the number of communication opera- tions per unit of work in the program, depends on many programming factors we will see in Chapters 2 and 3) and many hardware design factors. In par hardware may limit the transfer size and thereby determine the minimum numbe messages. It may automatically replicate data or migrate it to where it is ever, a certain amount of communication is inherent to parallel execution s must be shared and processors must coordinate their work. In general, for a to support programs with a high communication frequency, the other parts o communication cost equation must be small—low overhead, low network delay, a small occupancy. The attention paid to communication costs essentially dete which programming models a machine can realize efficiently and what portio the application space it can support. Any parallel computer with good co tional performance can support programs that communicate infrequently, but frequency or volume of communication increase, greater stress is pla munication architecture.

The *overlap* is the portion of the communication operation that is per currently with other useful work, including computation or other cc This reduction of the effective cost is possible because much of th time involves work done by components of the system other than th such as the communication assist, the bus, the network, or the remo memory. Overlapping communication with other work is a form of smal allelism, as is the instruction-level parallelism exploited by fast effect, we may invest some of the available parallelism in a proc actual cost of communication.

1.3.5 Summary

The issues of naming, operation set, and ordering apply at each lev in a parallel architecture, not just the programming model. In ge translation or run-time software may intervene between the progran and the communication abstraction, and beneath this abstraction are abstractions. At any level, communication and replication are deeply ever two processes access the same data, the data either needs to b between the two or replicated so each can access a copy of it. The a same name refer to two distinct physical locations in a meaningful m level of abstraction depends on the position adopted on naming and c level. Wherever data movement is involved, we need to understand it characteristics in terms of latency and bandwidth and, furthermore, influenced by overhead and occupancy. As architects, we need to desi frequency and type of operations that occur at the communication understanding that trade-offs occur across this boundary, involvi ported directly in hardware and what is supported in software. The p on naming, operation set, and ordering at each of these levels h 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 evo puter architecture, rooted essentially in the beginnings of computi this history it takes on a novel, even exotic role as the avenue fo and beyond what the base technology can provide. Parallel computer demonstrated a rich diversity of structure, usually motivated by spe parallel programming models. However, the dominant technological f VLSI generation have pushed parallelism increasingly into the mains parallel architecture almost ubiquitous. All modern microprocessors allel internally, executing several bit-parallel instructions in e reordering instructions within the limits of inherent dependences

costs of communication with hardware components external to the processor i
These microprocessors have become the performance and price-performance lea
of the computer industry. From the most powerful supercomputers to departme
servers to the desktop, we see systems constructed by utilizing multiples
processors integrated into a communications fabric. This technological fo
increasing maturity of compiler technology, has brought about a dramatic c
gence in the structural organization of modern parallel machines. The key a
tural issue is how communication is integrated into the memory and I/O sy
that form the remainder of the computational node. This communications arch
ture reveals itself functionally in terms of what can be named at the hardw
what ordering guarantees are provided, and how synchronization operations a
formed whereas, from a performance point of view, we must understand the inh
latency and bandwidth of the available communication operations. Thus, mo
parallel computer architecture carries with it a strong engineering compone
nable to quantitative analysis of cost and performance trade-offs.

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

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

Chapter 2 provides an overview of parallel programs—what they look like
how they are constructed. Chapter 3 explains the issues that must be addre
the programmer and compiler to construct a "good" parallel program, that i
that is effective enough in using multiple processors to form a reasonable
architectural evaluation. Ultimately, we design parallel computers against
gram characteristics at the machine level, so the goal of Chapter 3 is to c
nection between what appears in the program text and how the machine spend
time. In effect, Chapters 2 and 3 take us from a general understanding of
the application level to a specific understanding of the character and fre
operations at the communication abstraction level.

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

machine in isolation and then measuring how well they perform coll understanding of application characteristics is important to ensur load run on the machine stresses the various aspects of interest. S process for evaluating hypothetical architectural advancements. I which no machine exists need to be evaluated through simulations, w severe restrictions on what can reasonably be executed. Again, an u application characteristics and how they scale with problem and mach cial to navigating the design space.

Chapters 5 and 6 study in detail the design of symmetric multipro shared physical address space. Going deeply into the small-scale cas ing scalable designs is important for several reasons. First, small sors are the most prevalent form of parallel architecture; they ar form most students are exposed to, most software developers are targ professional designers are dealing with. Second, the issues that a scale are indicative of what is critical in the large scale, but t simpler and easier to grasp. Thus, these chapters provide a study what the following five chapters address in the large. Third, the s processor design is a fundamental building block for the larger-sca available options for interfacing a scalable interconnect with a p node are largely circumscribed by the processor, cache, and memory s small-scale machines. Finally, the solutions to key design problem scale case are elegant in their own right.

The fundamental building block for the designs in Chapters 5 and 6 bus between processors and memory. The basic problem that we need t keep the contents of the caches coherent and the view of memory pr processors consistent. A bus is a powerful mechanism. It provid communication through a single set of wires; moreover, it can serve medium, since there is only one set of wires, and even provide gl wired-OR signals. The properties of bus transactions are exploite extensions of conventional cache controllers that solve the cohe Chapter 5 presents the fundamental techniques for bus-based cache co logical level and presents the basic design alternatives. These d provide an illustration of how workload-driven evaluation can be bro making design decisions. Finally, Chapter 5 examines the parallel issues of the earlier chapters in terms of the aspects of machine de software level, especially with regard to cache effects on sharing design of robust synchronization routines. Chapter 6 focuses on the structure and machine implementation of bus-based cache coherence. I variety of more advanced designs that seek to reduce latency and width while preserving a consistent view of memory.

Chapters 7 through 11 form a closely interlocking study of the de parallel architectures. Chapter 7 makes the conceptual step from a b a building block for higher-level abstractions to a network transac block. To cement this understanding, the communication abstractions surveyed in this introductory chapter are constructed from primitive networ actions. Then the chapter studies the design of the node-to-network inte depth using a spectrum of case studies.

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

Chapter 10 takes up the question of the design of the scalable network i with processors, caches, and memory systems, the network design space has s dimensions, and often a design decision involves interactions along these sions. The chapter lays out the fundamental design issues for scalable inte illustrates the common design choices, and evaluates them relative to the ments established in Chapters 8 and 9. Chapter 11 draws together the materi the previous four chapters in the context of techniques for latency toleran ing bulk transfer, write behind, and read ahead across the spectrum of comm tion abstractions. Finally, Chapter 12 looks at the overall concepts of t light of technological, application, and economic trends and forecasts the ing developments in the field of parallel computer architecture.

1.5 HISTORICAL REFERENCES

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

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

Through the late 1960s, tremendous innovation occurred in the use
ism within the processor using pipelining and replication of functio
a far greater range of performance within a family than could be ob
increasing the clock rate. It was argued that these efforts were r
diminishing returns, so the University of Illinois and Burroughs un
research project to design and build a 64-processor SIMD machine, c
(Bouknight et al. 1972), based on the earlier Solomon work (and
Amdahl's arguments to the contrary [Amdahl 1967]). This project was
tious, involving research in the basic hardware technologies, a
devices, operating systems, programming languages, and applications
scaled-down, 16-processor system was working in 1975, the computer
undergone massive structural change.

First, the concept of storage as a simple linear array of moderat
devices had been revolutionized, beginning with the idea of virtu
then with the concept of caching. Work on Multics and its predecess
and CTSS) separated the concept of the user address space from the
ory of the machine. This required maintaining a short list of rece
translation lookaside buffer (TLB), in order to obtain reasonable pe
rice Wilkes, the designer of EDSAC, saw this as a powerful technique
the addressable storage itself, giving rise to what we now call the
an interesting example of locality triumphing over parallelism. The
caches into the 360/85 yielded higher performance than the 360/91,
faster clock rate, faster memory, and elaborate pipelined instructi
dynamic scheduling. The use of caches was commercialized in the IBM
this raised a serious difficulty for the I/O controllers as well as t
sors. If addresses were cached and therefore not bound to a part
location, how was an access from another processor or controller to
data? One solution was to maintain a directory of the location of ea
idea that has regained importance in recent years.

Second, storage technology itself underwent a revolution with s
memories replacing core memories. Initially, this technology was mos
small cache memories. Other machines, such as the CDC 7600, simply
separate, small, fast, explicitly addressed memory. Third, integra
hold. The combined result was that uniprocessor systems enjoyed
advance in performance, which mitigated much of the added value of
the Illiac IV system, with its inferior technological and architect
vector processing in the CDC STAR-100 addressed the class of numeri
tions that Illiac was intended to solve but eliminated the difficul
operations. The final straw was the introduction of the CRAY-1 sys
astounding 80-MHz clock rate owing to exquisite circuit design and t
we now call a RISC instruction set, augmented with vector operation
registers and offering high peak rate with very low start-up cost.
vector processing coupled with fast, expensive ECL circuits was to
performance computing for the next 15 years.

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

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

Simultaneously, the message-passing direction took off with two major re efforts. At CalTech, a project was started to construct a 64-processor sys i8086/8087 microprocessors assembled in a hypercube configuration (Seitz 1 Athas and Seitz 1988). From this baseline, several other designs were pur CalTech and JPL (Fox et al. 1988), and at least two companies pushed the ap into commercialization—Intel, with the iPSC series, and Ametek. A somewhat aggressive approach was widely promoted by the INMOS Corporation in England the form of the Transputer, which integrated four communication channels di onto the microprocessor. This approach was adopted by nCUBE, with a serie very large-scale message-passing machines. Intel carried the commodity pro approach forward, replacing the i80386 with the faster i860, then replacing work with a fast grid-based interconnect in the Delta and adding dedicated processors in the Paragon. Meiko moved away from the Transputer to the i8 their computing surface. IBM also investigated an i860-based des before obtaining commercial success with the SP family, essential RS6000 workstations.

Data parallel systems also took off in the early 1980s, after a quiet. These included Batcher's MPP system for image processing Goodyear and the Connection Machine promoted by Hillis for AI appli lis 1985). The key enhancement was the provision of a general-purpos for problems demanding other than simple grid-based communication. saw commercialization with the emergence of Thinking Machines Corpo with the CM-1, which was close to Hillis's original conceptions, ar CM-2, which incorporated a large number of bit-parallel floating-p addition, MasPar and Wavetracer carried the bit-serial or slightly w forward in cost-effective systems.

A more formal development of highly regular parallel systems em early 1980s as systolic arrays, generally under the assumption that very simple processing elements would fit on a single chip. It was these arrays would provide cheap, high-performance, special-purpos conventional computer systems. To some extent, these ideas have beer programming data parallel machines. The iWARP project at CMU produc general, smaller-scale building block that has been developed furthe with Intel. These ideas have also found their way into fast graphi and rendering chips.

The technological possibilities of the VLSI revolution also promp gation of more radical architectural concepts, including dataflow arc nis 1980; Gurd, Kerkham, and Watson 1985; Papadopoulos and Culler 19 and Culler 1986), which integrated the network very closely with t scheduling mechanism of the processor. It was argued that very fast uling throughout the machine would hide the long communication later chronization costs of a large machine and thereby vastly simplify pr evolution of these ideas tended to converge with the evolution of architectures in the form of message-driven computation (Dally, Kee 1993).

Large-scale shared memory designs took off as well. IBM pursued a research effort with the RP-3 (Pfister et al. 1985), which sought t number of early RISC processors (the 801) through a butterfly netwo based on the NYU Ultracomputer work (Gottlieb et al. 1983), which larly novel for its use of combining operations. BBN developed t designs, the BBN Butterfly using Motorola 68000 processors and the T Beranek and Newman 1989) using the 88100s. These efforts prompted a investigation of the possibility of providing cache-coherent shar scalable setting. The DASH project at Stanford University sought to cache-coherent distributed shared memory by maintaining a director the disposition of every cache block (Lenoski et al. 1993; Lenoski represented an effort to standardize an interconnect and cache coh

(IEEE 1993). The Alewife project at MIT sought to minimize the hardware su for shared memory (Agarwal et al. 1995), which was pushed further by resea at the University of Wisconsin (Wood et al. 1993). The Kendall Square Res KSR1 (Frank, Burkhardt, and Rothnie 1993; Saavedra, Gains, and Carlton 1 went even further and allowed the home location of data in memory to mig Alternatively, the Denelcor HEP attempted to hide the cost of remote me latency by interleaving many independent threads on each processor.

The 1990s have exhibited the beginnings of a dramatic convergence among t various factions. This convergence is driven by many factors. One is clear of the approaches have common requirements. They all require a fast, high- interconnect. They all profit from avoiding latency where possible and reduc absolute latency when it does occur. They all benefit from hiding as much communication cost as possible. They all must support various forms of sync zation. We have seen the shared memory work explicitly seek to better in message passing in Alewife (Agarwal et al. 1995) and FLASH (Kuskin et al. 1 obtain better performance where the regularity of the application can prov transfers. We have seen data parallel designs incorporate complete commodit cessors in the CM-5 (Leiserson et al. 1996), allowing very simple processin sages at the user level, which provides much better efficiency for message computing and shared memory (von Eicken et al. 1992; Spertus et al. 1993). remains the additional support for fast global synchronization. We have s global synchronization, message queues, and latency-hiding techniques deve in a NUMA shared memory context in the CRAY T3D (Kessler and Schwarzmeier 1993; Koeninger, Furtney, and Walker 1994), and the message-passing suppor the Meiko CS-2 (Barton, Crownie, and McLaren 1994; Homewood and McLaren 1993) provides direct virtual-memory-to-virtual-memory transfers within th address space. The new element that continues to separate the factions is t complete commodity workstation nodes, as in the SP-1, SP-2, and various w station clusters using merging high-bandwidth networks (Anderson, Culler, Patterson 1995; Kung et al. 1989; Pfister 1995). The costs of weaker integra the memory system, imperfect network reliability, and general-purpose s requirements have tended to keep these systems more closely aligned with tional 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 by fitting an exponential to the technology leaders using the data in Ta Obtain more recent data from the Web, and see how well these trends have he

1.2 Compute the annual performance growth rates for each of the benchmarks s 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 improve
performance, or speedup, due to some enhancement. Formally,

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

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

Suppose you are given a program that does a fixed amount of work, and sc
fraction s of that work must be done sequentially. The remaining portion of
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].) I
why it is an upper bound.

1.4 Given a histogram of available parallelism such as that shown in Figure
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
produce the speedup curve shown in that figure.

1.5 Locate the current TPC performance data on the Web and compare the mix o
tem configurations, performance, and speedups obtained on those machines w
the data presented in Figure 1.4.

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

1.7 To move an n-byte message along H links in an unloaded store-and-forward net-
work 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 t $\frac{n}{W}$ +
$(H-1)R$. Consider an 8×8 grid consisting of 40-MB/s links and routers with 25(
ns of delay. What is the minimum, maximum, and average time to move a 64-b
message through the network? A 256-byte message?

1.8 Consider a simple 2D finite difference scheme where at each step every poi
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 pe
cessor 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 a
minimize the data traffic. Compute how much data must be communicated pe
step.

1.9 Consider the simple pipelined component described in Example 1.2
the application alternates between n independent operations on the com-
ponent and phases of computation lasting T that do not use the component.
Develop an expression describing the execution time of the program i
parameters. Compare this with the unpipelined and fully pipelined bo
points do you get the maximum discrepancy between the models? How la
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 ti
actually transmit the data across the bus. The start-up includes t
access to the bus, convey the address, access the memory, and poss
data in the cache before responding to the processor. However, in a
sor with dynamic instruction scheduling, the overhead may include
tion spent accessing the cache to detect the miss and placing the re
The memory access portion contributes to latency, which can potenti
by the overlap with execution of instructions that do not depend o
the load.

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

1.13 Suppose this 32-byte line is transferred to another processor an
tion architecture imposes a start-up cost of 2 μs and data transfer bandwidth o
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
same model as in Exercise 1.12. The start-up can be thought of as
zero-length message; it includes the software overhead on the two
cost of accessing the network interface, and the time to actually c
The transfer time is usually determined by the point along the pat
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 pe
obtained?

1.15 In some cases, Equation 1.6 can be used for estimating data tran
based on design parameters. In other cases, it serves as an empiric
measurements to a line to determine the effective start-up and peak
portion of a system. If data undergoes a series of copies as part o
ing that before transmitting a message the data must be copied int
basic message time is as in Exercise 1.14, but the copy is perform
cycles per 32-bit word on a 100-MHz machine. Given an equation for
user-level message time, how does the cost of a copy compare with a
say, entering the operating system?

1.16 Consider a machine running at 100 MIPS on some workload with the follc mix: 50% ALU, 20% loads, 10% stores, and 20% branches. Suppose the instruc miss rate is 1%, the data miss rate is 5%, and the cache line size is 32 by purpose of this calculation, treat a store miss as requiring two cache lin one to load the newly updated line and one to replace the dirty line. If tl provides a 250-MB/s bus, how many processors can it accommodate at 50% of p 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 5 room on the bus to make the calculation reasonable. As the bus approaches tion, however, it takes longer to obtain access for the bus, so it loc processor as if the memory system is slower. The effect is to slow down a processors in the system, thereby reducing their bandwidth demand. Let's analogous calculation from the other direction.

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

1.18 Of course, no matter how many processors you place on the bus, they wil exceed the available bandwidth. Explain what happens to processor performan response to bus contention. Can you formalize your observations?