Chapter 1

Introduction

Jack Dongarra, Ken Kennedy and Andy White

"Nothing you can't spell will ever work." Will Rogers

1.1 Parallel Computing

Parallel computing is more than just a strategy for achieving high performance — it is a compelling vision for how computation can seamlessly scale from a single processor to virtually limitless computing power. This vision is decades old, but it was not until the late 1980s that it seemed within our grasp. However, the road to scalable parallelism has been a rocky one and, as of the writing of this book, parallel computing cannot be viewed as an unqualified success.

True, parallel computing has made it possible for the peak speeds of high-end supercomputers to grow at a rate that exceeded Moore's law. Unfortunately, the scaling of application performance has not matched the scaling of peak speed, and the programming burden for these machines continues to be heavy. This is particularly problematic because the vision of seamless scalability cannot be achieved without having the applications scale automatically as the number of processors increases. However, for this to happen, the applications have to be programmed to be able to exploit parallelism in the most efficient possible way. Thus the responsibility for achieving the vision of scalable parallelism falls on the application developer.

The Center for Research on Parallel Computation was founded in 1989 with the goal of making parallel programming easy enough so that it would be accessible to ordinary scientists. To do this, the Center conducted research on software and algorithms that could form the underpinnings of an infrastructure for parallel programming. The result of much of this research was captured in software systems and published algorithms, so that it could be widely used in the scientific community. However, the published work has never been collected into a single resource and, even if it had been, it would not incorporate the

broader work of the parallel computing research community.

This book is an attempt to fill that gap. It represents the collected knowledge of and experience with parallel computing from a broad collection of leading parallel computing researchers, both within and outside of CRPC. It attempts to provide both tutorial material and more detailed documentation of advanced strategies produced by research over the last two decades.

In the remainder of this chapter we delve more deeply into three key aspects of parallel computation — hardware, applications, and software — to provide a foundation and motivation for the material that will be elaborated later in the book. We begin with a discussion of the progress in parallel computing hardware. This is followed by a discussion of what we have learned from the many application efforts that were focused on exploitation of parallelism. Finally, we briefly discuss the state of parallel computing software and the prospects for such software in the future. We conclude with some thoughts about how close we are to a true science of parallel computation.

1.2 Parallel Computing Hardware

In last 50 years, the field of scientific computing has undergone rapid change — we have experienced a remarkable turnover of vendors, architectures, technologies and the usage of systems. Despite all these changes, the long-term evolution of performance seems to be steady and continuous, following the famous Moore's Law rather closely. In Figure 1.1, we plot the peak performance over the last five decades of computers that could have been called "supercomputers." This chart shows clearly how well this Moore's Law has held over almost the complete lifespan of modern computing — we see an increase in performance averaging two orders of magnitude every decade.

In the second half of the seventies, the introduction of vector computer systems marked the beginning of modern supercomputing. These systems offered a performance advantage of at least one order of magnitude over conventional systems of that time. Raw performance was the main, if not the only, selling point for supercomputers of this variety. However, in the first half of the eighties the integration of vector systems into conventional computing environments became more important. Only those manufacturers that provided standard programming environments, operating systems and key applications were successful in getting the industrial customers that became essential for survival in the marketplace. Performance was increased primarily by improved chip technologies and by producing shared-memory multiprocessor systems.

Fostered by several government programs, scalable parallel computing using distributed memory became the focus of interest at the end of the eighties. Overcoming the hardware scalability limitations of shared memory was the main goal of these new systems. The increase of performance of standard microprocessors after the RISC revolution, together with the cost advantage of large-scale parallelism, formed the basis for the "Attack of the Killer Micros" [143]. The transition from ECL to CMOS chip technology and the usage of "off the shelf" microprocessors instead of custom processors for MPPs was the consequence.

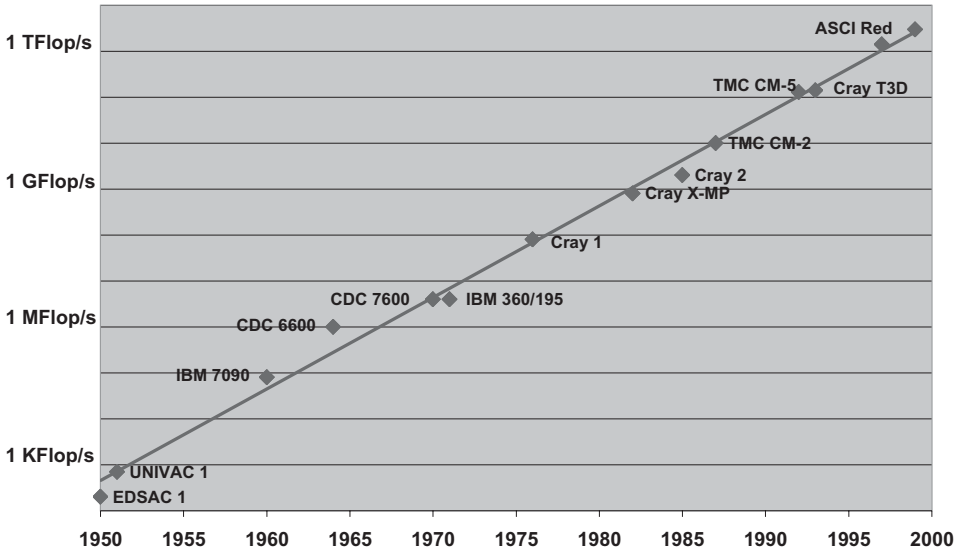

Figure 1.1: **Moore's Law and Peak Performance of Various Computers Over Time.**

At the beginning of the nineties, while the MP vector systems reached their widest distribution, a new generation of MPP systems came on the market, claiming to equal or even surpass the performance of vector MPs. To provide a more reliable basis for statistics on high-performance computers, the Top500 [287] list was begun. This report lists the sites that have the 500 most powerful installed computer systems. The best LINPACK benchmark performance [282] achieved is used as a performance measure to rank the computers. The TOP500 list has been updated twice a year since June 1993. In the first Top500 list in June 1993, there were already 156 MPP and SIMD systems present (31% of the total 500 systems).

The year 1995 saw remarkable changes in the distribution of the systems in the Top500 according to customer types (academic sites, research labs, industrial/commercial users, vendor installations, and confidential sites). Until June 1995, the trend in the Top500 data was a steady decrease of industrial customers, matched by an increase in the number of government-funded research sites. This trend reflects the influence of governmental HPC programs that made it possible for research sites to buy parallel systems, especially systems with distributed memory. Industry was understandably reluctant to follow this path, since systems with distributed memory have often been far from mature or stable. Hence, industrial customers stayed with their older vector systems,

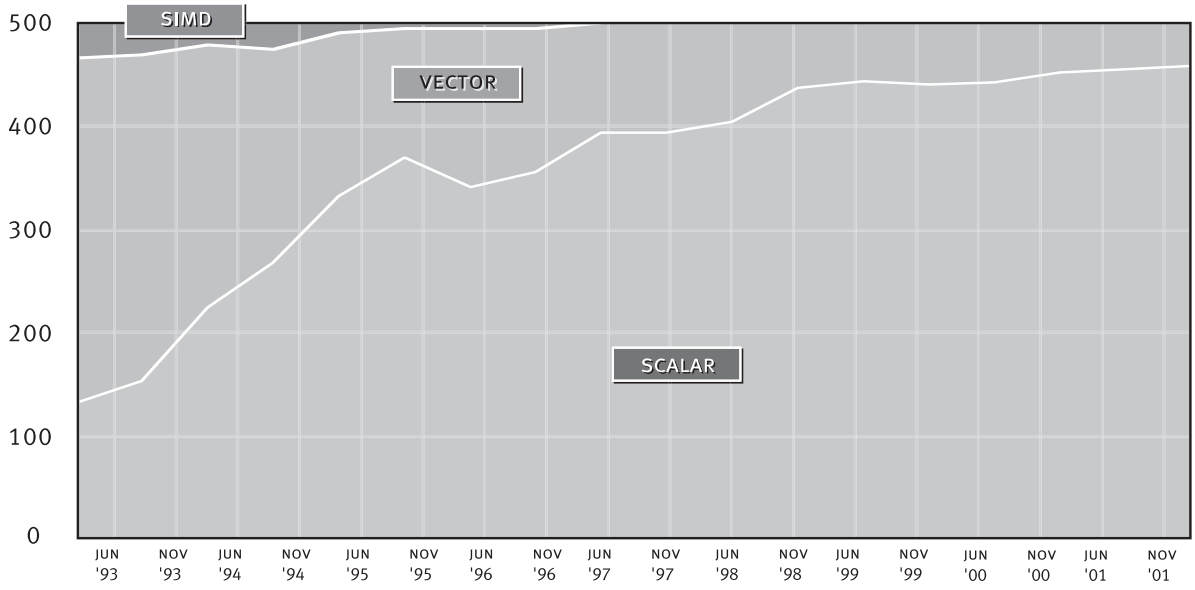

Figure 1.2: **Processor Design Used as Seen in the Top500.**

which gradually dropped off the Top500 list because of low performance.

Beginning in 1994, however, companies such as SGI, Digital, and Sun began selling symmetric multiprocessor (SMP) models in their workstation families. From the very beginning, these systems were popular with industrial customers because of the maturity of the architecture and their superior price/performance ratio. At the same time, IBM SP2 systems began to appear at a reasonable number of industrial sites. While the SP was initially intended for numerically intensive applications, in the second half of 1995 the system began selling successfully to a larger commercial market, with dedicated database systems representing a particularly important component of sales.

It is instructive to compare the growth rates of the performance of machines at fixed positions in the Top 500 list with those predicted by Moore's Law. To make this comparison, we separate the influence from the increasing processor performance and from the increasing number of processors per system on the total accumulated performance. (To get meaningful numbers we exclude the SIMD systems for this analysis, as they tend to have extremely high processor numbers and extremely low processor performance.) In Figure 1.3 we plot the relative growth of the total number of processors and of the average processor performance, defined as the ratio of total accumulated performance to the number of processors. We find that these two factors contribute almost equally to the annual total performance growth — a factor of 1.82. On average, then

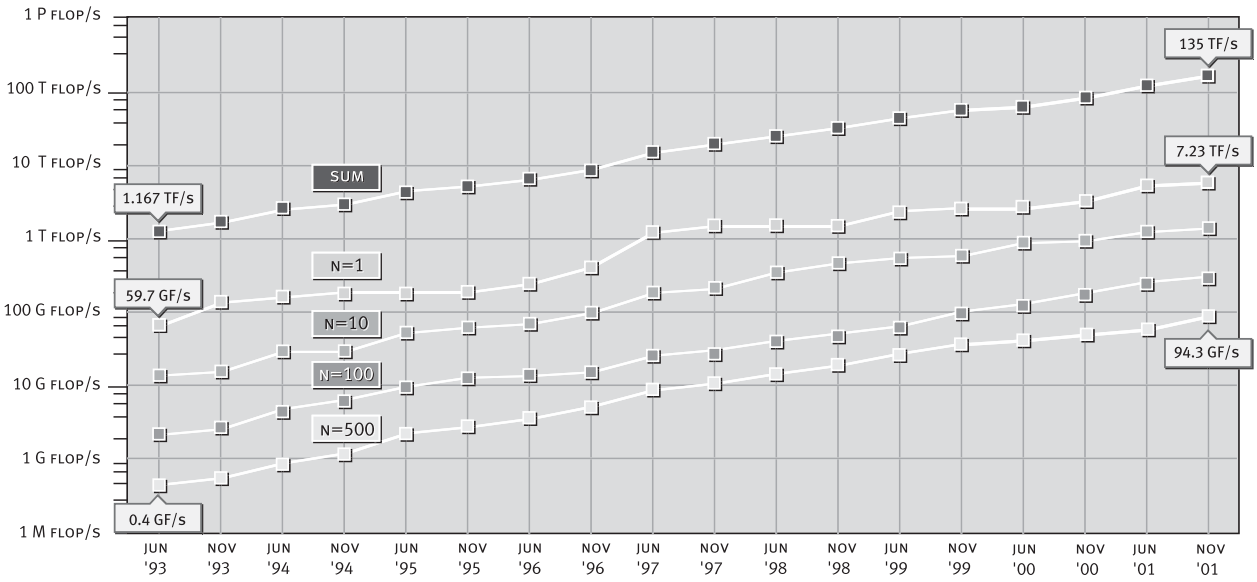

Figure 1.3: **Performance Growth at Fixed Top500 Rankings.**

number of processors grows by a factor of 1.30 each year and the processor performance by a factor 1.40 per year, compared to the factor of 1.58 predicted by Moore's Law.

Based on the current Top500 data (which cover the last 6 years) and the assumption that the current rate of performance improvement will continue for some time to come, we can extrapolate the observed performance and compare these values with the goals of government programs such as High Performance Computing and Communications and the PetaOps initiative. In Figure 1.4 we extrapolate the observed performance using linear regression on a logarithmic scale. This means that we fit exponential growth to all levels of performance in the Top500. This simple curve fit of the data shows surprisingly consistent results. Based on the extrapolation from these fits, we can expect to see the first 100 TFlop/s system by 2005, which is about 1–2 years later than the original ASCI projections. By 2005, no system smaller then 1 TFlop/s should be able to make the Top500 any more.

Looking even further in the future, we speculate that based on the current doubling of performance every year the first PetaFlop/s system should be available around 2009. Due to the rapid changes in the technologies used in HPC systems, there is currently no reasonable projection possible for the architecture of the PetaFlops systems at the end of the decade. Even as the HPC market has changed substantially since the introduction of the Cray 1 three decades

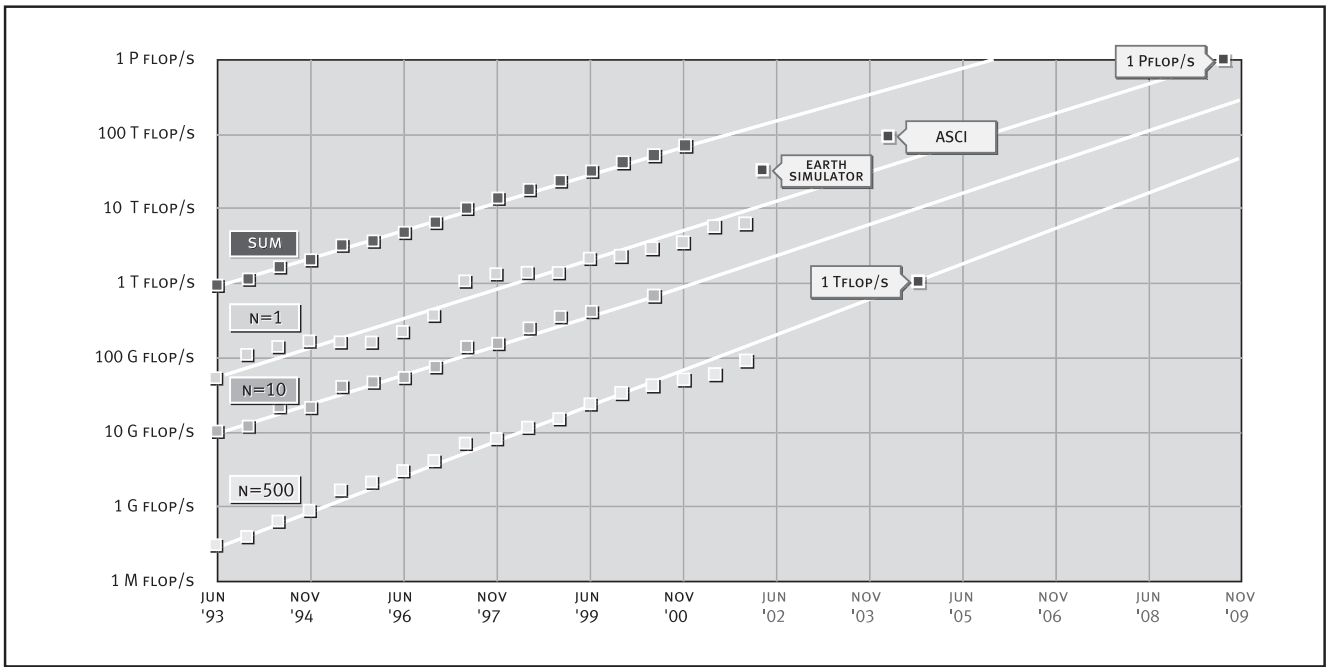

Figure 1.4: **Extrapolation of Top500 Results.**

ago, there is no end in sight for these rapid cycles of architectural redefinition.

There are two general conclusions we can draw from these figures. First, parallel computing is here to stay. It is the primary mechanism by which computer performance can keep up with the predictions of Moore's law in the face of the increasing influence of performance bottlenecks in conventional processors. Second, the architecture of high-performance computing will continue to evolve at a rapid rate. Thus, it will be increasingly important to find ways to support scalable parallel programming without sacrificing portability. This challenge must be met by the development of software systems and algorithms that promote portability while easing the burden of program design and implementation.

1.3 What Have We Learned from Applications?

Remarkable strides have been taken over the last decade in utilization of high-end, that is parallel, computers. Federal agencies, most notably NSF, DOE, NASA, and DoD have provided increasingly powerful, scalable resources to scientists and engineers across the country. We will discuss a handful of lessons learned that punctuate the lifetime of the CRPC and provide important context for the next millennium.

Parallel computing can transform science and engineering. Scalable, parallel computing has transformed a number of science and engineering disciplines, in-

cluding cosmology, environmental modeling, condensed matter physics, protein folding, quantum chromodynamics, device and semiconductor simulation, seismology, and turbulence [450]. As an example, consider cosmology [729] — the study of the universe, its evolution and structure — where one of the most striking paradigm shifts has occurred. A number of new, tremendously detailed observations deep into the universe are available from such instruments as the Hubble Space Telescope and the Digital Sky Survey. However, until recently, it has been difficult, except in relatively simple circumstances, to tease from mathematical theories of the early universe enough information to allow comparison with observations.

However, scalable parallel computers with large memories have changed all of that. Now, cosmologists can simulate the principal physical processes at work in the early universe over space-time volumes sufficiently large to determine the large-scale structures predicted by the models. With such tools, some theories can be discarded as being incompatible with the observations. High-performance computing has allowed comparison of theory with observation and thus has transformed the practice of cosmology.

To port or not to port. That is not the question. "Porting" a code to parallel architectures is more than simply bringing up an existing code on a new machine. Because parallel machines are fundamentally different from their vector predecessors, porting presents an opportunity to reformulate the basic code and data structures and, more importantly, to reassess the basic representation of the processes or dynamics involved. As an example, consider ocean modeling, where the standard Bryan–Cox–Semtner (BCS) code was retargeted from Cray vector architecture to the CM-2 and CM-5 [877]. The BCS model was inefficient in parallel for two reasons: the primary loop structure needed to be reorganized, and global communications were required by the stream-function formulation of the BCS representation. The latter feature of the BCS model required that independent line integrals be performed around each island in the calculation. The model was reformulated in surface-pressure form, where the solution of the resulting equations does not require line integrals around islands and is better conditioned than the mathematically equivalent stream-function representation. An additional change to the original model relaxed the "rigid-lid" approximation that suppressed surface-gravity waves (and allowed longer time steps) in the BCS model.

In addition to a more efficient code on either a vector or parallel architecture, this reformulation brought several remarkable benefits:

1. islands were treated with simple, pointwise boundary conditions, thereby allowing all island features to be included at a particular resolution;

2. unsmoothed bottom topography could be used without adverse effects on convergence; and

3. a free-surface boundary at the ocean–air interface made the sea-surface

height a prognostic variable and allowed direct comparison with Topex–Poseidon satellite altimeter data.

Satellite data has become a key feature in the verification and validation of global ocean models [105].

Parallel supercomputing can answer challenges to society. Computational science has just begun to make an impact on problems with direct human interest and on systems whose principal actors are not particles and aggregations of particles, but rather are humans and collections of humans.

Perhaps the most oft-asked and rarely-answered question about scientific computing concerns predicting the weather. However, there are some things that can be said. Hurricane tracks are being more accurately predicted [564], which directly reduces the cost of evacuations and which indirectly reduces loss of life. This increased fidelity is equal parts computation and observation — more accurate and detailed data on hurricane wind-fields is available using dropwindsondes that report not only the meteorological variables of interest, but also an accurate position by using GPS. Another significant development over the last decade has been the Advanced Regional Prediction System [828] developed by the NSF Science and Technology Center for the Analysis and Prediction of Storms `[www.caps.out.edu]`.

However, basically this work still concerns modeling of physical systems, in this case severe weather, which have significant impact on society. A more difficult job is effectively modeling society itself, or a piece thereof. For example, detailed environmental impact statements are required prior to any significant change in metropolitan transportation systems. In order to meet this regulatory requirement, the Department of Transportation commissioned development of a transportation model. The result, TRANSIMS [944], models traffic flow by representing all of the traffic infrastructure (e.g. streets, freeways, lights, stop signs), developing a statistically consistent route plan for the area's population, and then simulating the movement of each car, second by second. The principal distinction here is that we believe that precise mathematical laws exist that accurately characterize the dynamics and interplay of physical systems. No such systems of laws, with the possible exception of Murphy's, is contemplated for human-dominated systems.

It's not the hardware, stupid. The focus has often been on computing hardware. The reasons are straightforward: they cost a lot of money and take a lot of time to acquire; they have measurable, often mysterious except to the fully initiated, properties; and we wonder how close they are getting to the most famous computer of all, HAL. However, if we contrast the decade of the Crays to the tumultuous days of the MPPs, we realize that it was the consistency of the programming model, not the intricacies of the hardware, that made the former 'good old' and the latter 'interesting'.

A case in point is seismic processing [247]. Schlumberger acquired two, 128-node CM-5s to provide seismic processing services to their customers. They were successful simply because it was possible, in this instance, to write an efficient

post-stack migration code for the CM-5 and provide commercial quality services to their customers, all within the 2-4 year operational window of any given high-end hardware platform. Those programs or businesses that could not profitably, or possibly, write new applications for each new hardware system were forced to continue in the old ways. However, the Schlumberger experience teaches us an important lesson: a successful high-end computing technology must have a stable, effective programming model that persists over the lifetime of the application. In the case of Stockpile Stewardship, this is on the order of a decade.

In conclusion, applications have taught us much over the last ten years.

1. Entire disciplines can move to firm scientific foundation by using scalable, parallel computing to expand and elucidate mathematical theories, thus allowing comparison with observation and experiment.

2. High-end computing is beginning to make an impact on everyday life, on society, by providing more accurate, detailed, and trusted forecasts and predictions, even on human-dominated systems.

3. New approaches to familiar problems, taken in order to access high capacity, large memory parallel computers, can have tremendous ancillary benefits beyond mere restructuring of the computations.

4. A persistent programming model for scalable, parallel computers is absolutely essential if computational science and engineering is to realize even a fraction of its remarkable promise.

5. The increase in the accuracy, detail, and volume of observational data goes hand in hand with these same improvements in the computational arena.

1.4 Software and Algorithms

As we indicated at the beginning of this chapter, the widespread acceptance of parallel computation has been impeded by the difficulty of the parallel programming task. First, the expression of an explicitly parallel program is difficult — in addition to specifying the computation and how it is to be partitioned among processors, the developer must specify the synchronization and data movement needed to ensure that the program computes the correct answers and achieves high performance.

Second, because the nature of high-end computing systems changes rapidly, it must be possible to express programs in a reasonably machine-independent way, so that moving to new platforms from old ones is possible with a relatively small amount of effort. In other words, parallel programs should be portable between different architectures. However, this is a difficult ideal to achieve because the price of portability is often performance.

The goal of parallel computing software systems should be to make parallel programming easier and the resulting applications more portable while achieving the highest possible performance. This is clearly a tall order.

A final complicating factor for parallel computing is the complexity of the problems being attacked. This complexity requires extraordinary skill on the part of the application developer along with extraordinary flexibility in the developed applications. Often this means that parallel programs will be developed using multiple programming paradigms and often multiple languages. Interoperability is thus an important consideration in choosing the development language for a particular application component.

The principal goal of the Center for Research on Parallel Computation (CRPC) has been the development of software and algorithms that address programmability, portability, and flexibility of parallel applications. Much of the material in this book is devoted to the explication of technologies developed in CRPC and the community to ameliorate these problems. These technologies include new language standards and language processors, libraries that encapsulate major algorithmic advances, and tools to assist in the formulation and debugging of parallel applications.

In the process of carrying out this research we have learned a number of hard but valuable lessons. These lessons are detailed in the next few paragraphs.

Portability is elusive. When CRPC began, every vendor of parallel systems offered a different application programming interface. This made it extremely difficult for developers of parallel applications because the work of converting an application to a parallel computer would need to be repeated for each new parallel architecture. One of the most important contributions of CRPC was an effort to establish cross-platform standards for parallel programming. The MPI and HPF standards are just two results of this effort.

However, portability is not just a matter of implementing a standard interface. In scientific computing most users are interested in *portable performance*, which means the ability to achieve a high fraction of the performance possible on each machine from the same program image. Because the implementations of standard interfaces are not the same on each platform, portability even for programs written in MPI or HPF has not been automatically achieved. Typically the implementor must spend significant amounts of time tuning an application for each new platform.

This tuning burden even extends to programming via portable libraries, such as ScaLAPACK. Here the CRPC approach has been to isolate the key performance issues in a few kernels that could be rewritten by hand for each new platform. Still the process remains tedious.

Algorithms are not always portable. An issue impacting portability is that an algorithm does not always work well on every machine architecture. The differences arise because of number and granularity of processors, connectivity and bandwidth, and the performance of the memory hierarchy on each individual processor. In many cases, portable algorithm libraries must be parameterized to do algorithm selection based on the architecture on which the individual routines are to run. This makes portable programming even more difficult.

Parallelism isn't everything. One of the big surprises on parallel computers was the extent to which poor performance arises because of factors other than insufficient parallelism. The principal problem on scalable machines other than parallelism is data movement. Thus, the optimization of data movement between processors is a critical factor in performance of these machines. If this is not done well, a parallel application is likely to run poorly no matter how powerful the individual processors are. A second and increasingly important issue affecting performance is the bandwidth from main memory on a single processor. Many parallel machines use processors that have so little bandwidth relative to the processor power that the processor cycle time could be dialed down by a factor of two without affecting the running time of most applications. Thus as parallelism levels have increased, algorithms and software have had to increasingly deal with memory hierarchy issues, which are now fundamental to parallel programming.

Community acceptance is essential to the success of software. Technical excellence alone cannot guarantee that a new software approach will be successful. The scientific community is generally conservative, in the sense that they will not risk their effort on software strategies that are likely to fail. To achieve widespread use, there has to be the expectation that a software system will survive the test of time. Standards are an important part of this, but cannot alone guarantee success. A case in point is High Performance Fortran (HPF). In spite of the generally acknowledged value of the idea of distribution-based languages and a CRPC-led standardization effort, HPF failed to achieve the level of acceptance of MPI because the commercial compilers did not mature in time to gain the confidence of the community.

Good commercial software is rare at the high end. Because of the small size of the high-end supercomputing market, commercial software production is difficult to sustain unless it also supports a much broader market for medium-level systems, such as symmetric (shared-memory) multiprocessors. OpenMP has succeeded because it targets that market, while HPF was focused on the high end. The most obvious victim of market pressures at the high end are tools — tuners and debuggers — which are usually left until last by the vendors and often abandoned. This has seriously impeded the widespread acceptance of scalable parallelism and has led to a number of community-based efforts to fill the gap based on open software. Some of these efforts are described in later chapters.

1.5 Toward a Science of Parallel Computation

When the Center for Research on Parallel Computation began activities in 1989, parallel computing was still in its infancy. Most commercial offerings consisted of a few processors that shared a single memory. Scalable parallel systems, though long a subject of research, had just graduated from laboratories to become prototype commercial products. Almost every parallel computer had a different proprietary programming interface. Since that time, many machines (and companies) have come and gone.

To deal with the rapid pace of change in parallel systems, application developers needed principles and tools that would survive in the long term and isolate them from the changing nature of the underlying hardware. On the other hand, they also needed new parallel algorithms and an understanding of how to match them to different architectures with differing numbers of processors. In short, they needed a science of parallel computation.

Fostering such a science was a major goal of the High Performance Computing and Communications (HPCC) initiative, launched by the Federal government in 1991. With the help of this initiative, CRPC and the entire parallel computing research community have made major strides toward the desired goal. Message-passing interfaces have been standardized; a variety of higher-level programming interfaces have been developed and incorporated into commercial products; debugging systems and I/O interfaces have matured into useful standards; and many new parallel algorithms have been developed and incorporated into widely distributed libraries and templates.

Why, in the face of these advances, is the science of parallel computation still interesting to study? When CRPC began its activities in 1989, many of us felt that we could develop a higher-level parallel programming interface that would supplant the message-passing paradigms then being used. However, our expectation that explicit message passing would routinely be hidden from the developer has not been realized. Today, most developers must use explicit message passing, albeit via a more sophisticated portable interface, to generate efficient scalable parallel programs. This is but one example demonstrating that the science of parallel computation is incomplete.

It is possible that our original goal was unrealistic and that the desired science cannot be achieved. We doubt this, as we now understand much better the demands of applications and intricacies of high performance architectures. We understand better where parallel performance is essential and where the developer needs programming paradigms optimized more for functionality than performance; this corresponds to the emerging picture of hybrid systems as a Grid of loosely coupled high performance parallel "kernels". Furthermore, we now have better technologies than those available a decade ago. These technologies come not only from deeper understanding of the problem of parallel computing but also from new ideas. These include the broader acceptance of object-based languages, powerful scripting environments, and the growing understanding of the role of meta-data in defining how computation should be applied dynamically. Java, Python and the Semantic Web are illustrative of technologies reflecting these new ideas.

Our intent is that this book document the current science of parallel computation, including the best methods, algorithms, and tools that were developed during the 11 years of CRPC's activities, and thus serve as a useful resource for practicing application developers. In addition, we hope it will motivate new approaches to the support of parallel programming that could lead finally to the realization of our original dream.

Chapter 2

Parallel Computer Architectures

William Gropp, Rick Stevens, and Charlie Catlett

Parallel computers provide great amounts of computing power, but they do so at the cost of increased difficulty in programming and using them. Certainly, a uniprocessor that was fast enough would be simpler to use. To explain why parallel computers are inevitable and to identify the challenges facing developers of parallel algorithms, programming models, and systems, in this chapter we provide an overview of the architecture of both uniprocessor and parallel computers. We will show that while computing power can be increased by adding processing units, memory latency (the irreducible time to access data) is the source of many challenges in both uniprocessor and parallel processor design. In Chapter 3, "Parallel Programming Considerations," some of these issues will be revisited from the perspective of the programming challenges they present.

Parallel architectures and programming models are not independent. While most architectures can support all major programming models, they may not be able to do so with enough efficiency to be effective. An important part of any parallel architecture is any feature that simplifies the process of building (including compiling), testing, and tuning an application. Some parallel architectures put a great deal of effort into supporting a parallel programming model; others provide little or no extra support. All architectures represent a compromise between cost, complexity, timeliness, and performance. Chapter 12 discusses some of the issues of parallel languages and compilers; for parallel computers that directly support parallel languages, the ability to compile efficient code is just as important as it is for the single-processor case.

This chapter is organized as follows. In Section 2.1 we briefly describe the important features of single processor (or uniprocessor) architecture. From this background, the basics of parallel architecture are presented in Section 2.2; in particular, we describe the opportunities for performance improvement through

parallelism at each level in a parallel computer, with references to machines of each type. In Section 2.3, we examine potential future parallel computer architectures. We conclude the chapter with a brief summary of the key issues motivating the development of parallel algorithms and programming models.

We note that this chapter discusses only parallel architectures used in high-performance computing. Parallelism is widely used in commercial computing for applications such as databases and Web servers. Special architectures and hardware have been developed to support these applications, including special hardware support for synchronization and fault tolerance.

2.1 Uniprocessor Architecture

In this section we briefly describe the major components of a conventional, single-processor computer, emphasizing the design tradeoffs faced by the hardware architect. This description lays the groundwork for a discussion of parallel architectures, since parallelism is entirely a response to the difficulty of obtaining ever greater performance (or reliability) in a system that inherently performs only one task at a time. Readers familiar with uniprocessor architecture may skip to the next section. Those interested in a more detailed discussion of these issues should consult [466].

The major components of a computer are the central processing unit (CPU) that executes programs, the memory system that stores executing programs and the data that the programs are operating on, and input/output systems that allow the computer to communicate with the outside world (e.g., through keyboards, networks, and displays) and with permanent storage devices such as disks. The design of a computer reflects the available technology; constraints such as power consumption, physical size, cost, and maintainability; the imagination of the architect; and the software (programs) that will run on the computer (including compatibility issues). All of these have changed tremendously over the past fifty years.

Perhaps the best-known change is captured by Moore's law [679], which says that microprocessor CPU performance doubles roughly every eighteen months. This is equivalent to a thousandfold increase in performance over fifteen years. Moore's law has been remarkably accurate over the past thirty-six years (see Figure 2.1), even though it represents an observation about (and a driver of) the rate of engineering progress and is not a law of nature (such as the speed of light). In fact, it is interesting to look at the clock speed of the *fastest* machines in addition to (and compared with) that of microprocessors. In 1981, the Cray 1 was one of the fastest computers, with a 12.5 ns clock. In 2001, microprocessors with 0.8 ns clocks are becoming available. This is a factor of 16 in twenty years, or equivalently a doubling every five years.

Remarkable advances have occurred in other areas of computer technology as well. The cost per byte of storage, both in computer memory and in disk storage, has fallen along a similar exponential curve, as has the physical size per byte of storage (in fact, cost and size are closely related). Dramatic advancements in algorithms have reduced the amount of work needed to solve many classes of

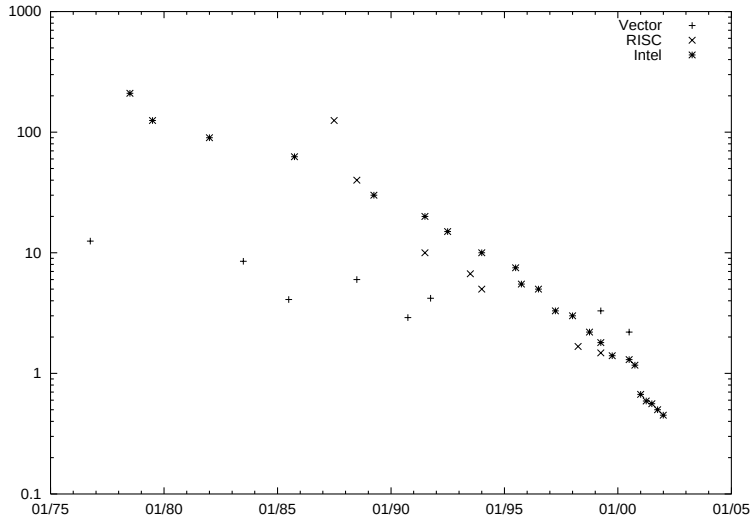

Figure 2.1: Improvement in CPU performance measured by clock rate.

important problems; for example, the work needed to solve n simultaneous linear equations has fallen, in many cases, from n^3 to n. For one million equations, this is an improvement of 12 orders of magnitude!

Unfortunately, these changes have not been uniform. For example, while the density of storage (memory and disk) and the bandwidths have increased dramatically, the decrease in the time to access storage (latency) has not kept up. As a result, over the years, the balance in performance between the different parts of a computer has changed. In the case of storage, increases in clock rates relative to storage latency have translated Moore's law into a description of *inflation* in terms of the relative cost of memory access from the point of view of potentially wasted CPU cycles. This has forced computer architectures to evolve over the years, for example moving to deeper and more complex memory hierarchies.

2.1.1 The CPU

The CPU is the heart of the computer; it is responsible for all calculations and for controlling or supervising the other parts of the computer. A typical CPU contains the following (see Figure 2.2):

Arithmetic Logic Unit (ALU): Performs computations such as addition and comparison.

Floating Point Unit (FPU): Performs operations on floating-point numbers.

Load/Store Unit: Performs loads and stores for data.

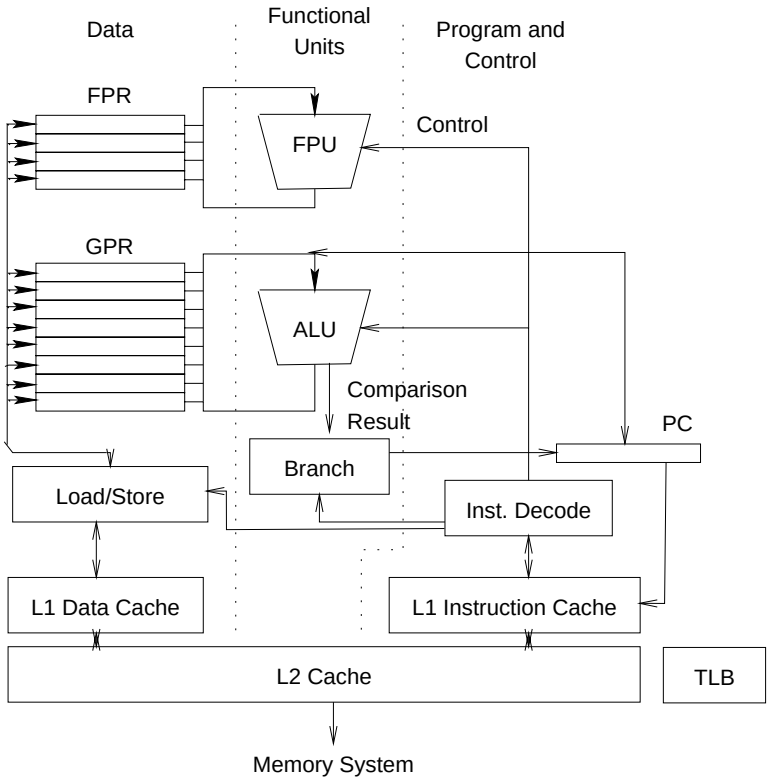

Figure 2.2: Generic CPU diagram. This example has a separate L1 cache for data and for program instructions and a unified (both data and instructions) L2 cache. Not all data paths are shown.

Registers: Fast memory locations used to store intermediate results. These are often subdivided into floating-point registers (FPRs) and general purpose registers (GPRs).

Program Counter (PC): Contains the address of the instruction that is executing.

Memory Interface: Provides access to the memory system. In addition, the CPU chip often contains the fastest part of the memory hierarchy (the top-level cache); this part is described in Section 2.1.2.

Other components of a CPU are needed for a complete system, but the ones listed are most important for our purpose.

The CPU operates in steps controlled by a clock: in each step, or *clock cycle*, the CPU performs an operation.[1] The speed of the CPU clock has increased

[1]Note that we did not say an instruction or a statement. As we will see, modern CPUs may perform both less than an instruction and more than one instruction in a clock cycle.

dramatically; desktop computers now come with clocks that run at over 2 GHz (2×10^9 Hz).

One of the first decisions that a computer architect must make is what basic operations can be performed by the CPU. There are two major camps: the complex instruction set computer (CISC) and the reduced instruction set computer (RISC). A RISC CPU can do just as much as a CISC CPU; however, it may require more instructions to perform the same operation. The tradeoff is that a RISC CPU, because the instructions are fewer and simpler, may be able to execute each instruction faster (i.e., the CPU can have a higher clock speed), allowing it to complete the operation more quickly.

The specific set of instructions that a CPU can perform is called the instruction set. The design of that instruction set relative to the CPU represents the instruction set architecture (ISA). The instructions are usually produced by compilers from programs written in higher-level languages such as Fortran or C. The success of the personal computer has made the Intel x86 ISA the most common ISA, but many others exist, particularly for enterprise and technical computing. Because most programs are compiled, ISA features that either aid or impede compilation can have a major impact on the effectiveness of a processor. In fact, the ability of compilers to exploit the relative simplicity of RISC systems was a major reason for their development.

We note that while the ISA may be directly executed by the CPU, another possibility is to design the CPU to convert each instruction into a sequence of one or more "micro" instructions. This allows a computer architect to take advantage of simple operations to raise the "core" speed of a CPU, even for an ISA with complex instructions (i.e., a CISC architecture). Thus, even though a CPU may have a clock speed of over 1 GHz, it may need multiple clock cycles to execute a single instruction in the ISA. Hence, simple clock speed comparisons between different architectures are deceptive. Even though one CPU may have a higher clock speed than another, it may also require more clock cycles than the "slower" CPU in order to execute a single instruction.

Programs executed by the CPU are stored in memory. The *program counter* specifies the address in memory of the executing instruction. This instruction is fetched from memory and decoded in the CPU. As each instruction is executed, the PC changes to the address of the next instruction. Control flow in a program (e.g., `if`, `while`, or function all) is implemented by setting the PC to a new address.

One important part of the ISA concerns how memory is accessed. When memory speeds were relatively fast compared with CPU speeds (particularly for complex operations such as floating-point division), the ISA might have included instructions that read several items from memory, performed the operation, and stored the result into memory. These were called memory-to-memory operations. However, as CPU speeds increased dramatically relative to memory access speeds, ISAs changed to emphasize a "load-store" architecture. In this approach, all operations are performed by using data in special, very fast locations called *registers* that are part of the CPU. Before a value from memory can be used, it must first be loaded into a register, using an address that

has been computed and placed into another register. Operations take operands from registers and put the result back into a register; these are sometimes called register-to-register operations. A separate store operation puts a value back into the memory (generally indirectly by way of a cache hierarchy analogous to the register scheme just described). Load and store operations are often handled by a load/store functional unit, much as floating-point arithmetic is handled by a floating-point unit (FPU).

Over the years, CPUs have provided special features to support various programming models. For example, CISC-style ISAs often include string search instructions and even polynomial evaluation. Some current ISAs support instructions that make it easy to access consecutive elements in memory by updating the register holding the load address; this corresponds closely to the `a=*x++;` statement in the C programming language and to typical Fortran coding practice for loops.

One source of complexity in a CPU is the difference in the complexity of the instructions. Some instructions, such as bitwise logical **or,** are easy to implement in hardware. Others, such as floating-point division, are extremely complicated. Memory references provide a different kind of complexity; as we will see, the CPU often cannot predict when a memory reference will complete. Many different approaches have been taken to address these issues. For example, in the case of floating-point operations, *pipelining* has been used. Like the RISC approach, pipelining breaks a complex operation into separate parts. Unlike the RISC approach, however, each stage in the pipeline can be executed at the same time by the CPU, but on different data. In other words, once a floating-point operation has been started in a clock cycle, even though that operation has not completed, a new floating-point operation can be started in the next clock cycle. It is not unusual for operations to take two to twenty cycles to complete. Figure 2.3 illustrates a pipeline for floating-point addition. Pipelines have been getting deeper (i.e., have more stages) as clock speeds increase. Note also that this hardware approach is very similar to the use of pipelining in algorithms described in Section 3.2.2. As CPUs have become faster, pipelining has been used more extensively. In modern CPUs, many other instructions (not just floating point operations) may be pipelined.

From this discussion, we can already see some of the barriers to achieving higher performance. A clock rate of 1 GHz corresponds to a period of only 1 ns. In 1 ns, light travels only about 1 foot in a vacuum, and less in an electrical circuit. Even in the best case, a single processor running at 10 GHz (three more doublings in CPU performance or, if Moore's law continues to hold, appearing in less than five years) and its memory could be only about one inch across (any larger and a signal could not cross the chip during a single clock cycle); at that size, heat dissipation becomes a major problem (in fact, heat dissipation is already a problem for many CPUs). Approaches such as pipelining (already a kind of parallelism) require that enough operations and operands be available to keep the pipeline full. Other approaches begin to introduce a very fine scale of parallelism, for example by providing multiple *functional units* such as multiple floating-point adders and multipliers. In such cases, however, the program must

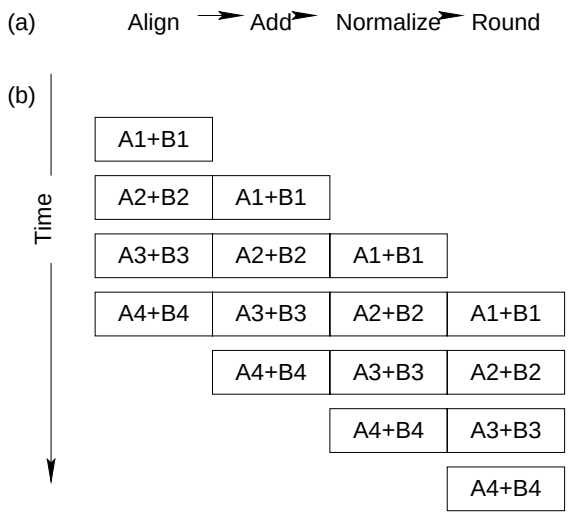

Figure 2.3: Example of a floating-point pipeline. The separate stages in the pipeline are shown in (a). In (b), four pairs of numbers are added in 7 clock cycles. Note that after a 3-cycle delay, one result is returned every cycle. Without pipelining, 16 clock cycles would be required to add four pairs of numbers.

be rewritten and/or recompiled to make use of the additional resources. (These enhancements are discussed in Section 2.2.3.)

Once on-chip clock latency is addressed, the designer must face an even more challenging problem: latency to storage, beginning with memory.

2.1.2 Memory

While a computer is running, active data and programs are stored in memory. Memory systems are quite complex, introducing a number of design issues. Among these are the following:

Memory size. Users never have enough computer memory, so the concept of *virtual memory* was introduced to fool programs into thinking that they have large amounts of memory just for their own use.

Memory latency and hierarchy. The time to access memory has not kept pace with CPU clock speeds. Levels or *hierarchies* of memory try to achieve a compromise between performance and cost.

Memory bandwidth. The rate at which memory can be transferred to and from the CPU (or other devices, such as disks) also has not kept up with CPU speeds.

Memory protection. Many architectures include hardware support for memory protection, aimed primarily at preventing application software from mod-

ifying (intentionally or inadvertently) either system memory or memory in use by other programs.

Of these, memory latency is the most difficult problem. Memory size, in many ways, is simply a matter of money. Bandwidth can be increased by increasing the number of paths to memory (another use of parallelism) and using techniques such as interleaving. Latencies are related to physical constraints and are harder to reduce. Further, high latencies reduce the effective bandwidth of a given load or store. To see this, consider a memory interconnect that transfers blocks of 32 bytes with a bandwidth of 1 GB/s. In other words, the time to transfer 32 bytes is 32 ns. If the latency of the memory system is also 32 ns (an optimistic figure), the total time to transfer the data is 64 ns, reducing the effective bandwidth from 1 GB/s to 500 MB/s. The most common approach to improving bandwidth in the presence of high latency is to increase the amount of data moved each time, thus amortizing the latency over more data. However, this helps only when all of the data moved is needed by the running program. Chapter 3 discusses this issue in detail from the viewpoint of software.

An executing program, or *process*, involves an address space and one or more program counters. Operating systems manage the time-sharing of a CPU to allow many processes to appear to be running at the same time (for parallel computers, the processes may in fact be running simultaneously). The operating system, working with the memory system hardware, provides each process with the appearance of a private address space. Most systems further allow the private memory space to appear larger than the available amount of physical memory. This is called a *virtual address space*. Of course, the actual physical memory hardware defines an address space, or *physical address space*. Any memory reference made by a process, for example, with a load or store instruction, must first be translated from the virtual address (the address known to the process) to the physical address. This step is performed by the *translation lookaside buffer* (TLB), which is part of the memory system hardware. In most systems, the TLB can map only a subset of the virtual addresses (it is a kind of address cache); if a virtual address can't be handled by the TLB, the operating system is asked to help out; in such a case, the cost of accessing memory greatly increases. For this reason, some high-performance systems have chosen not to provide virtual addressing.

Finding ways to decrease memory latency is a difficult problem. To understand why, we must first look at how computer memory works. Semiconductor memory comes in two main types: static random access memory (SRAM), in which each bit of memory is stored in a latch made up of transistors, and dynamic random access memory (DRAM), in which each bit of memory is stored as a charge on a capacitor. SRAM is faster than DRAM but is much less dense (has fewer bits per chip) and requires much greater power (resulting in heat). The difference is so great that virtually all computers use DRAM for the majority of their memory. However, as Figure 2.4 shows, the performance of DRAM memory has not followed the Moore's law curve that CPU clock speeds have. Instead, the density and price-performance of DRAMs have risen exponentially.

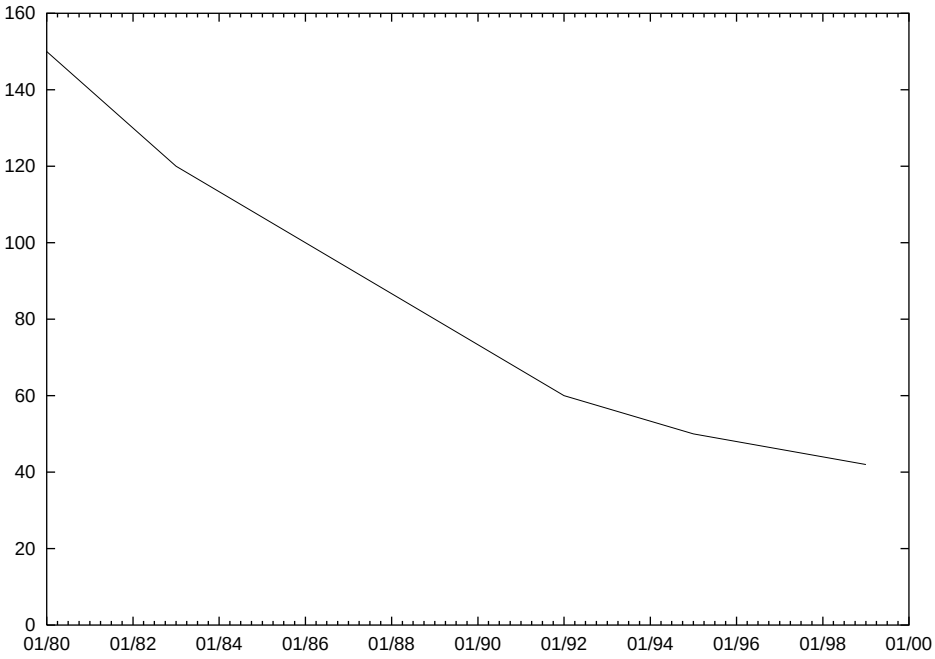

Figure 2.4: DRAM latency versus time. Note that, unlike the CPU times in Figure 2.1, the time axis is linear, and the improvement in performance is little more than a factor of two in ten years.

The scale of this problem can be seen by comparing the speeds of DRAMs and CPUs. For example, a 1 GHz CPU will execute 60 instructions before a typical (60 ns) DRAM can return a single byte. Hence, in a program that issues a load for a data item that must come from DRAM, at least 60 cycles will pass before the data will be available. In practice, the delay can be longer because there is more involved in providing the data item than just accessing the DRAM.

To work around this performance gap, computer architects have introduced a hierarchy of smaller but faster memories. These are called *cache memories* because they work by caching copies of data from the DRAM memory in faster SRAM memory, closer to the CPU. Because SRAM memory is more expensive and less dense and consumes much more power than does DRAM memory, cache memory sizes are small relative to *main memory*. In fact, there is usually a hierarchy of cache memory, starting from level 1 (L1), which is the smallest (and fastest) and is on-chip on all modern CPUs. Many systems have two or three levels of cache. A typical size is 16 KB to 128 KB for L1 cache memory to as much as 4 MB to 8 MB for L2 or L3 cache memory. Typical DRAM memory sizes, on the other hand, are 256 MB to 4 GB—a factor of about a thousand larger.

Memory hierarchy brings up another problem. Because the cache memory is so much smaller than the main memory, it often isn't possible for all of the memory used by a process to reside in the L1 or even L2 cache memory. Thus, as a process runs, the memory system hardware must decide which memory locations to copy into cache. If the cache is full and a new memory location is needed, some other item must be removed from the cache (and written back[2] to the main memory if necessary). If the CPU makes a request for data, and the requested data is not in cache, a *cache miss* occurs. The rate at which this happens is called the *cache miss rate*, and one of the primary goals of a memory system architect is to make the miss rate as small as possible. Of course, the rate depends on the behavior of the program, and this in turn depends on the algorithms used by the program. Many different strategies are used to try to achieve low miss rates in a cache while keeping the cache fast and relatively inexpensive. To reduce the miss rate, programs exploit *temporal locality*: reusing the same data within a short span of time, that is, reusing the data before it is removed from the cache to make room for some other data. This process, in turn, requires the algorithm developer and programmer to pay close attention to how data is used in a program.

As just one example, consider the choice of the *cache line size*. Data between cache and main memory usually is transferred in groups of 64, 128, or 256 bytes. This group is called a cache line. Moving an entire cache line at one time allows the main memory to provide relatively efficient bursts of data (it will be at least 60 ns before we can get the first byte; subsequent consecutive bytes can be delivered without much delay). Thus, programs that access "nearby" memory after the first access will find that the data they need is already in cache. For these programs, a larger line size will improve performance. However, programs that access memory in a less structured way may find that they spend most of their time reading data into cache that is never used. For these programs, a large line size reduces performance compared with a system that uses a shorter cache line. Chapter 3 discusses these issues in more detail, along with strategies for reducing the impact of memory hierarchies on performance.

Many other issues also remain, with similarly difficult tradeoffs, such as associativity (how main memory addresses are mapped into the cache), replacement policy (what data is ejected to make room for new data), and cache size. Exploiting the fact that memory is loaded in larger units than the natural scalar objects (such as integers, characters, or floating-point numbers) is called *exploiting spatial locality*. Spatial locality also requires temporal locality.

The effective use of cache memory is so important for high-performance applications that algorithms have been developed tailored to the requirements of these memory hierarchies. On the other hand, the most widely used programming models ignore cache memory requirements. Hence, problems remain with the practical programming of these systems for high performance. We will also see in Section 2.2.1 that the use of copies of data in a cache causes problems for

[2]Write-back caches wait until an item is displaced from the cache to write the data back into memory. Write through is another approach that stores to memory at the same time as the data is stored into the cache. Other approaches can be used as well.

parallel systems.

In recent years, there has been rapid progress in memory system design, particularly for personal computers and workstations. Many of these designs have focused on delivering greater bandwidth and have names like RAMBUS, DDR (for double data rate), and EDO. See [240] and other articles in the same issue for a discussion of high-performance DRAM technologies.

2.1.3 I/O and Networking

Discussions of computers often slight the issues of I/O and networking. I/O, particularly to the disks that store files and swap space for supporting virtual memory, has followed a path similar to that of main memory. That is, densities and sizes have increased enormously (twenty-five years ago, a 40 MB disk was large and expensive; today, a 40 GB disk is a commodity consumer item), but latencies have remained relatively unchanged. Because disks are electromechanical devices, latencies are in the range of milliseconds or a million times greater than CPU speeds. To address this issue, some of the same techniques used for memory have been adopted, particularly the use of caches (typically using DRAM memory) to improve performance.

Networking has changed less. Although Ethernet was introduced twenty-two years ago, only relatively modest improvements in performance were seen for many years, and most of the improvement has been in reduced monetary cost. Fortunately, in the past few years, this situation has started to change. In particular, 100 Mb Ethernet has nearly displaced the original 10 Mb Ethernet, and several Gigabit networking technologies are gaining ground, as are industry efforts, such as Infiniband [507], to accelerate the rate of improvement in network bandwidth. Optical technologies have been in use for some time but are now poised to significantly increase the available bandwidths. Networks, are, however, fundamentally constrained by the speed of light. Latencies can never be less than three nanoseconds per meter. Another constraint is the way in which the network is used by the software. The approaches that are currently used by most software involve the operating system (OS) in most networking operations, including most data transfers between the main memory and the network. Involving the OS significantly impacts performance; in many cases, data must be moved several times. Recent developments in networking [966, 968] have emphasized transfers that are executed without the involvement of the operating system, variously called "user-mode," "OS bypass," or "scheduled transfer." These combine hardware support with a programming model that allows higher network performance.

2.1.4 Summary

The design of a single-processor computer is a constant struggle against competing constraints. How should resources be allocated? Is it better to use transistors on a CPU chip to provide a larger fast L1 cache, or should they be used to improve the performance of some of the floating-point instructions? Should transistors be used to add more functional units? Should there be more

registers, even if the ISA then has to change? Should the L1 cache be made larger at the expense of the L2 cache? Should the memory system be optimized for applications that make regular or irregular memory accesses? There are no easy answers here. The complexity has in fact led to increasingly complex CPU designs that use tens or even hundreds of millions of transistors and that are enormously costly to design and manufacture. Particularly difficult is the mismatch in performance between memory and CPU. This mismatch also causes problems for programmers; see, for example, [537] for a discussion of what should be a simple operation (bit reversal) but whose performance varies widely as a result of the use of caches and TLBs. These difficulties have encouraged computer architects to consider a wide variety of alternative approaches for improving computer system performance. Parallelism is one of the most powerful and most widely used.

2.2 Parallel Architectures

This section presents an overview of parallel architectures, considered as responses to limitations and problems in uniprocessor architectures and to technology opportunities. We start by considering parallelism in the memory systems, since the choices here have the most effect on programming models and algorithms. Parallelism in the CPU is discussed next; after increases in clock rates, this is a source of much of the improvement in sustained performance in microprocessors. For a much more detailed discussion of parallel computer architectures, see [236].

2.2.1 Memory Parallelism

One of the easiest ways to improve performance of a computer system is simply to replicate entire computers and add a way for the separate computers to communicate data. This approach is shown schematically in Figure 2.5. This provides an easy way to increase memory bandwidth and aggregate processing power without changing the CPU, allowing parallel computers to take advantage of the huge investment in commodity microprocessor CPUs. The cost is in increased complexity of the software and in the impact that this has on the performance of applications. The major choice here is between distributed memory and shared memory.

Distributed Memory

The simplest approach from the hardware perspective is the *distributed-memory*, or *shared-nothing*, model. The approach here is to use separate computers connected by a network. The typical programming model consists of separate processes on each computer communicating by sending messages (*message passing*), usually by calling library routines. This is the most classic form of parallel computing, dating back to when the computers were people with calculators and the messages were written on slips of paper [803]. The modern distributed-memory parallel computer started with the work of Seitz [847].

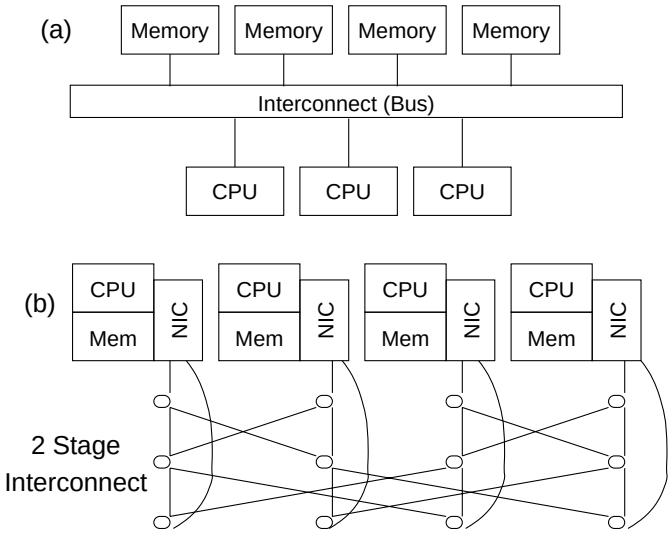

Figure 2.5: Schematic parallel computer organization. A typical shared-memory system is shown in (a), where the interconnect may be either a simple bus or a sophisticated switch. A distributed memory system is shown in (b); this may be either a distributed shared-memory system or a simpler shared-nothing system, depending on the capabilities of the network interface (NIC).

Distributed-memory systems are the most common parallel computers because they are the easiest to assemble. Systems from Intel, particularly the Paragon and the 512-processor Delta, were important in demonstrating that applications could make effective use of large numbers of processors. Perhaps the most successful commercial distributed-memory system is the IBM SP family. SP systems combine various versions of the successful RS6000 workstation and server nodes with different interconnects to provide a wide variety of parallel systems, from 8 processors to the 8192-processor ASCI White system. Some distributed-memory systems have been built with special-purpose hardware that provides remote memory operations such as put and get. The most successful of these are the Cray T3D and T3E systems.

Many groups have exploited the low cost and relatively high performance of commodity microprocessors to build clusters of personal computers or workstations. Early versions of these were built from desktop workstations and were sometimes referred to as NOWs, for networks of workstations. The continued improvement in performance of personal computers, combined with the emergence of open source (and free) versions of the Unix operation system, gave rise to clusters of machines. These systems are now widely known as Beowulfs or Beowulf clusters, from a project begun by Thomas Sterling and Donald Becker at NASA [901, 902]. They are real parallel machines; as of 2000, two of the top 100 supercomputer systems were built from commodity parts.

We note that the term *cluster* can be applied both broadly (any system built with a significant number of commodity components) or narrowly (only commodity components and open-source software). In fact, there is no precise definition of a cluster. Some of the issues that are used to argue that a system is a massively parallel processor (MPP) instead of a cluster include proprietary interconnects (different interconnects are described in Section 2.2.2), particularly ones designed for a specific parallel computer, and special software that treats the entire system as a single machine, particularly for the system administrators. Clusters may be built from personal computers or workstations (either single processor or symmetric multiprocessors (SMP)) and may run either open-source or proprietary operating systems.

While the message-passing programming model has been successful, it emphasizes that the parallel computer is a collection of separate computers.

Shared Memory

A more complex approach ties the computers more closely together by placing all of the memory into a single (physical) address space and supporting virtual address spaces across all of the memory. That is, data is available to all of the CPUs through the load and store instructions of the ISA. Because access to the memory is through load and store operations rather than the network operations used in distributed-memory systems, access to remote memory has lower latency and higher bandwidth. These advantages come with a cost, however. The two major issues are *consistency* and *coherency*. The most serious problem (from the viewpoint of the programmer) is consistency. To understand this problem, consider the following simple Fortran program:

```
a = a + 1
b = 1
```

In a generic ISA, the part that increments the variable `a` might be translated into

```
...
LOAD R12, %A10  ;  Load a into register
ADD  R12, #1    ;  Add one to the value in R12
STORE R12, %A10 ;  Store the result back into A
...
```

The important point here is that the single program statement `a=a+1` turns into three separate instructions. Now, recall our discussion of cache memory. In a uniprocessor, the first time the `LOAD` operation occurs, the value is brought into the memory cache. The store operation writes the value from register *back into the cache*. Now, assume that another CPU, executing a program that is using the same address space, executes

```
10  if (b .eq. 0) goto 10
    print *, a
```

What value of `a` does that CPU see? We would like it to see the value of `a` after the increment. But that requires that the value has both been written back to the memory from the cache of the first CPU and read into cache (even if the corresponding cache line had previously been read into memory) on the second CPU. In other words, we want the program to execute as if the cache were not present, that is, as if every load and store operation worked directly on the memory and in the order in which they were written. The copies of the memory in the cache are used only to improve performance of memory operations; they do not change the behavior of programs that are accessing the same memory locations. Cache memory systems that accomplish this objective are called *cache coherent*. Most (but not all) shared memory systems are cache coherent. Ensuring that a memory system is cache coherent requires additional hardware and adds to the complexity of the system. On the other hand, it simplifies the job of the programmer, since the correctness of a program doesn't depend on details of the behavior of the cache. We will see, however, that while cache coherence is necessary, it is not sufficient to provide the programmer with a friendly programming environment.

The complexity of providing cache coherency has led to different designs. One important class is called *uniform memory access* (UMA). In this design, each memory and cache are connected to all of the others; each part observes any memory operation (such as a load from a memory location) and ensures that cache coherence is maintained. The name UMA derives from the fact that the time to access a location from memory (not from cache and on an unloaded or nearly idle machine) is independent of the address (and hence particular memory unit). Early implementations used a *bus*, which is a common signaling layer that connected each processor and memory. Because buses are not scalable (all devices on the bus must share a limited amount of communication), higher-performance UMA systems based on completely connected networks have been constructed. Such networks themselves are not scalable (the number of connections for p components grows as p^2), leading to another class of shared-memory designs.

The *nonuniform memory access* (NUMA) approach does not require that all memory be equally "distant" (in terms of access time). Instead, the memory may be connected by a scalable network. Such systems can be more sensitive to the details of data layout but can also scale to much larger numbers of processors. To emphasize that a NUMA system is cache coherent, the term CC-NUMA is often used. The term *distributed shared memory* (DSM) is also often used to emphasize the NUMA characteristics of this approach to building shared-memory hardware. The term *virtual shared memory*, or *virtual distributed shared memory*, is used to describe a system that provides the programmer with a shared-memory programming model built on top of distributed-memory (not DSM) hardware.

Shared-memory systems are becoming common, even for desktop systems. Most vendors include shared-memory systems among their offerings, including Compaq, HP, IBM, SGI, and Sun and many personal computer vendors. Most of these systems have between 2 and 16 processors; most of these are UMA

systems. Typical CC-NUMA systems include the SGI Origin 3000 (typically up to 128 processors, 1024 in special configurations) and the HP SuperDome (up to 64 processors). The SGI Origin uses an approach called directory-based cache coherency (directory caches, for short) [606] to distribute the information needed to maintain cache coherency across the network that connects the memory to the CPUs.

Shared-memory systems often have quite modest memory bandwidths. At the low end, in fact, the same aggregate memory bandwidth may be provided to systems with 1 to 4 or even 16 processors. As a result, some of these systems are often starved for memory bandwidth. This can be a problem for applications that do not fit in cache. Applications that are memory-access bound can even slow as processors are added in such systems. Of course, not all systems are underpowered, and the memory performance of even the low-end systems has been improving rapidly.

Memory Consistency and Programming Models

How does the programming model change when several threads or processes share memory? What are the new issues and concerns? Consider a uniprocessor CPU executing a single-user program (a single-threaded, single-process program). Programs execute simply, one statement after the other. Implicit in this is that all statements before the current statement have completed before the current statement is executed. In particular, all stores to and loads from memory issued by previous statements have completed before the current statement begins to execute. In a multiprocessor executing a single program on multiple processors, the notion of "current" statement and "completed before" is unclear — or rather, it can be defined to be clear, but only at a high cost in performance.

Section 3.2 discusses the question of when a program can be run in parallel and give the correct results. The discussion there focuses on the issues for software. Lamport [593] asked a similar question about the parallel computer hardware in an article titled "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs." From a programmer's perspective, a parallel program should execute as if it were some arbitrary interleaving (but preserving order) of the statements in the program. This requirement is called *sequential consistency* and is essentially a "what you see (or write) is what you get" requirement for executing parallel programs. Unfortunately, while this matches the way most programmers look at their code, it imposes severe constraints on the hardware, in large part because of the high latency of memory accesses relative to the CPU speed.

Because providing sequential consistency limits performance, weaker models have been proposed. One model proposed in the late 1980s, called *processor consistency* [393], matched many of the then-current multiprocessor implementations but (usually) required some explicit action by the programmer to ensure correct program behavior. Programmers who use the thread programming model with thread locks to synchronize accesses to shared data structures sat-

isfy this requirement because the implementation of the lock and unlock calls in the thread library ensures that the correct instructions are issued.

Some programmers prefer to avoid the use of locks, however, because of their relatively high overhead and instead use flag variables to control access to shared data (as we used `a` as the flag variable in the preceding section). *Weak consistency* [295] is appropriate for such programs; like processor consistency, the programmer is required to take special steps to ensure correct operation.

Even weak consistency interferes with some performance optimizations, however. For this reason, *release consistency* [378] was introduced. This form of consistency separates synchronization between two processes or threads into an *acquire* and a *release* step.

The important point for programmers and algorithm developers is that the programming model that is most natural for programmers and that reflects the way we read programs is sequential consistency, and this model is not implemented by parallel computer hardware. Consequently, the programmer cannot rely on programs executing as some interleaved ordering of the statements. The specific consistency model that is implemented by the hardware may require different degrees of additional specification by the programmer. Language design for parallel programming may take the consistency model into account, providing ways for the compiler, not the programmer, to enforce consistency. Unfortunately, most languages (including C, C++, and Fortran) were designed for single threads of control and do not provide any mechanism to enforce consistency.

Note that if memory latency was small, providing sequential consistency would not greatly impact performance. Weaker forms of consistency would not be needed, and Lamport's title [593] would reflect real machines. In addition, these concepts address only correctness of parallel programs. Chapter 3 discusses some of the performance issues that arise in parallel computers, such as the problem of *false sharing*. Section 2.2.5 describes some of the instruction-set features that are used by programming models to ensure correct operation of correct programs.

Other Approaches

Two other approaches to parallelism in memory are important. In both of these, the CPU is customized to work with the memory system. In single instruction, multiple data (SIMD) parallelism, simplified CPUs are connected to memory. Unlike the previous cases, in the SIMD approach, each CPU executes the *same* instruction in each clock cycle. Such systems are well suited for the *data-parallel* programming model, where data is divided up among memory systems and the same operation is performed on each data element. For example, the Fortran code

```
do i=1, 10000
    a(i) = a(i) + alpha * b(i)
enddo
```

can be converted into a small number of instructions, with each CPU taking a part of the arrays `a` and `b`. While these systems have fallen out of favor as general-purpose computers, they are still important in fields such as signal processing. The most famous general-purpose SIMD system was the Connection Machine (CM-1 and CM-2) [476].

The other major approach is vector computing. This is often not considered parallelism because the CPU has little explicit parallelism, but parallelism is used in the memory system. In vector computing, operations are performed on *vectors*, often groups of 64 floating-point numbers. A single instruction in a vector computer may cause 64 results to be computed (often with a pipelined floating-point unit), using vectors stored in vector registers. Data transfers from memory to vector registers make use of multiple memory *banks*; the parallelism in the memory supports very high bandwidths between the CPU and the memory. Vector computers often have memory bandwidths that are an order of magnitude or more greater than nonvector computers. We will come back to vector computing in Section 2.2.3 after discussing parallelism in the CPU.

Parallel vector processors represent one of the most powerful classes of parallel computer, combining impressive per processor performance with parallelism. As late as 1996, the top machines on the Top 500 list of supercomputers were parallel vector processors [940], and since then only massively parallel systems with thousands of processors are faster.

The fastest of these machines may not provide full cache coherency in hardware; instead, they may require some support from the software to maintain a consistent view of memory. Machines in this category include the NEC SX-5 and Cray SV1. This is an example of the sort of trade-off of performance versus cost and complexity that continues to face architects of parallel systems.

A distinguishing feature of vector processors and parallel vector processors is the high memory bandwidth, often 4–16 bytes per floating-point operation. This is reflected in the high sustained performance achieved on these machines for many scientific applications.

Parallel Random Access Memory

A great deal of theoretical work on the complexity of parallel computation has used the parallel random access memory model (PRAM). This is a theoretical model of a shared-memory computer; different varieties of PRAM vary in the details of how memory accesses by different threads or processes to the same address are handled. In order to make the theoretical model tractable, memory access times are usually considered constant independent of the CPU performing the (nonconflicting) access; in particular, there are no caches and no factors of one hundred or more difference in access times for different memory locations. While this model is valuable in understanding the limits of parallel algorithms, the PRAM model represents an abstraction that cannot be efficiently implemented in practice.

Limits to Memory System Performance

Latency can be hidden by issuing memory operations far enough ahead so that the data is available when needed. While hiding a few cycles of latency is possible, the large latencies to DRAM memory are difficult to hide. We can see this situation by applying Little's law to memory requests. Little's law is a result from queuing theory; applied to memory requests, it says that if the memory latency that needs to be hidden is L and the rate of requests is r, then the number of simultaneously active requests needed is rL. If this is cast in terms of clock cycles, if the memory latency is 100 cycles and a memory request is issued every cycle, then 100 requests must be active at the same time. The consequences include the following:

1. The bandwidth of the memory system must support more requests (the number uses the same formula but uses the latency of the interconnect, which may still be around 10 cycles).

2. There must be enough independent work. Some algorithms, particularly those that use recurrence relations, do not have much independent work. This situation places a burden on the algorithm developer and the programmer.

3. The compiler must convert the program into enough independent requests, and there must be enough resources (such as registers) to hold results as they arrive (load) or until they depart (store).

Many current microprocessors allow a small number of outstanding memory operations; only the Cray MTA satisfies the requirements of Little's law for main-memory accesses.

2.2.2 Interconnects

In the preceding section, we described the interaction of memories and CPUs. In this section we say a little more about the interconnection networks that are used to connect components in a computer (parallel or otherwise).

Many types of networks have been used in the past thirty years for constructing parallel systems, ranging from relatively simple buses, to 2-D and 3-D meshes, to Clos networks, and to complex hypercube network topologies [604]. Each type of network can be described by its topology, its means of dealing with congestion (e.g., blocking or nonblocking), its approach to message routing, and its bandwidth characteristics.

For a long time, understanding details of the topology was important for programmers and algorithm developers seeking to achieve high performance. This situation is reflected both in the literature and in parallel programming models (e.g., the topology routines in MPI). Recently, networks have improved to the point that for many users, network topology is no longer a major factor in performance. However, some of this apparent "flatness" (uniformity) in the topology comes from greatly increased bandwidth within the network. As

network endpoints become faster, network topology may again become an important consideration in algorithms and programming models. Congestion in the network can still be a problem if the network performance doesn't scale with the number of processing nodes. The term *bisection bandwidth* describes the bandwidth of the network across any cut that divides the network into two parts.

Note that there is no best approach. Simple mesh networks, such as those used in the Intel TFLOPS (ASCI Red) system, provide effective scalability for many applications through low latency and high bandwidth, even though a mesh network does not have scalable performance in the sense that the bisection bandwidth of a mesh does not grow proportionally with the number of nodes. It is scalable in terms of the hardware required: there is a constant cost per node for each node added.

When interconnects are viewed as networks between computers, the performance goals have been quite modest. Fast networks of this type typically have latencies of ten microseconds or more (including essential software overheads) and bandwidths on the order of 100 MB/s. Interconnects used to implement shared memory, on the other hand, are designed to operate at memory system speeds and with no extra software overhead. Latencies for these systems are measured in nanoseconds and bandwidths of one to ten gigabytes per second are becoming common.

Early shared-memory systems used a bus to connect memory and processors. A bus provides a single, shared connection that all devices use and is relatively inexpensive to build. The major drawback is that if k devices are using the bus at the same time, under the best of conditions, each gets $1/k$ of the available performance (e.g., bandwidth). Contention between devices on the bus can lower the available bandwidth considerably.

To address this problem, some shared-memory systems have chosen to use networks that connect each processor with each memory system. For small numbers of processors and memories, a direct connection between each processor and memory is possible (requiring p^2 connections for p devices); this is called a *full crossbar*. For larger numbers of processors, a less complete network may be used. A common approach is to build an interconnect out of several stages, each stage containing some number of full crossbars. This provides a complete interconnect at the cost of additional latency.

An interesting development is the convergence of the technology used for networking and for shared memory. The *scalable coherent interconnect* (SCI) [502] was an early attempt to provide a memory-oriented view of interconnects and has been used to build CC-NUMA systems from Hewlett-Packard. Building on work both in research and in industry, the VIA [966] and Infiniband [507] industry-standard interconnects allow data to be moved directly from one processor's memory to another along an established circuit. These provide a communication model that is much closer to that used in memory interconnects and should offer much lower latencies and higher bandwidths than older, message-oriented interconnects.

Systems without hardware-provided cache coherency often provide a way to

indicate that all copies of data in a cache should be discarded; this is called *cache invalidation*. Sometimes this is a separate instruction; sometimes it is a side effect of a synchronization instruction such as test-and-set (e.g., Cray SV-1). Software can use this strategy to ensure that programs operate correctly. The cost is that *all* copies of data in the cache are discarded; hence, subsequent operations that reference memory locations stall while the cache is refilled. To avoid this situation, some systems allow individual cache lines to be invalidated rather than the entire cache. However, such an approach requires great care by the software, since the failure to invalidate a line containing data that has been updated by another processor can lead to incorrect and nondeterministic behavior by the program.

For an engaging discussion of the challenges of implementing and programming shared-memory systems, see [762].

2.2.3 CPU Parallelism

Parallelism at the level of the CPU is more difficult to implement than simple replication of CPUs and memory, even when the memory presents a single shared address space. However, modest parallelism in the CPU provides the easiest route to improved performance for the majority of applications because little needs to be done by the programmer to exploit this kind of parallelism.

Superscalar Processing

Look at Figure 2.2 again, and consider the following program fragment:

```
real a, b, c
integer i, j, k
...
a = b * c
i = j + k
```

Assume that the values `a`, `b`, `c`, `i`, `j`, and `k` are already in register. These two statements use different functional units (FPU and ALU, respectively) and different register sets (FPR and GPR). A *superscalar* processor can execute both of these statements (each requiring a single register-to-register instruction) in the same clock cycle (more precisely, such a processor will "begin execution" of the two statements, since both may be pipelined). The term superscalar comes from the fact that more than one operation can be performed in a single clock cycle and that performance is achieved on nonvector code. A superscalar processor allows as much parallelism as there are functional units. Because separate instructions are executed in parallel, this is also called *instruction-level parallelism* (ILP). For ILP to be effective, it must be easy for the hardware to find instructions that do not depend on each other and that use different functional units. Consider the following example, where the CPU has one adder and one multiplier. If the CPU executes instructions in the order that they appear, then the code sequence on the left will take three cycles and the one on the right only two cycles.

```
a = b * c          a = b * c
d = e * f          i = j + k
i = j + k          d = e * f
l = m + n          l = m + n
```

Some CPUs will attempt to reorder instructions in the CPU's hardware, an action that is most beneficial to legacy applications that cannot be recompiled. It is often better, however, if the compiler *schedules* the instructions for effective use of ILP; for example, a good code-scheduling compiler would transform the code on the left to the code on the right (but breaking sequential consistency because the load/store order is not preserved!).

One major drawback of ILP, then, is that the hardware must rediscover what a scheduling compiler already knows about the instructions that can be executed in the same clock cycle.

Explicitly Parallel Instructions

Another approach is for the instruction set to encode the use of each part of the CPU. That is, each instruction contains explicit subinstructions for each of the different functional units in the CPU. Since each instruction must explicitly specify more details about what happens in each clock cycle, the resulting instructions are longer than in other ISAs. In fact, they are usually referred to as very long instruction word (VLIW) ISAs. VLIW systems usually rely on the compiler to schedule each functional unit. One of the earliest commercial VLIW machines was the Multiflow Trace. The Intel IA64 ISA is a descendent of this approach; the term EPIC (explicitly parallel instruction computing) is used for the Intel variety. EPIC does relax some of the restrictions of VLIW but still relies on the compiler to express most of the parallelism.

SIMD and Vectors

One approach to parallelism is to apply the same operation to several different data values, using multiple functional units. For example, a single instruction might cause four values to be added to four others, using four separate adders. We have seen this SIMD style of parallelism before, when applied to separate memory units. The SIMD approach is used in some current processors for special operations. For example, the Pentium III includes a small set of SIMD-style instructions for single-precision floating-point and related data move operations. These are designed for use in graphics transformations that involve matrix-vector multiplication by 4×4 matrices.

Vector computers use similar techniques in the CPU to achieve greater performance. A vector computer can apply the same operation to a collection of data called a vector; this is usually either successive words in memory or words separated by a constant offset or *stride*. Early systems such as the CDC Star 100 and Cyber 205 were vector memory-to-memory architectures where vectors could be nearly any length. Since the Cray 1, most vector computers have used vector registers, typically limiting vectors to 64 elements. The big advantage of vector computing comes from the regular memory access that a vector rep-

resents. Through the use of pipelining and other techniques such as chaining, a vector computer can completely hide the memory latency by overlapping the access to the next vector with operations on a current vector.

Vector computing is related to VLIW or explicitly parallel computing in the sense that each instruction can specify a large amount of work and that advanced compilers are needed to take advantage of the hardware. Vectors are less flexible than the VLIW or EPIC approach but, because of the greater regularity, can sustain higher performance on applications that can be expressed in terms of vectors.

Multithreading

Parallelism in the CPU involves executing multiple sets of instructions. Any one of these sets, along with the related virtual address space and any state, is called a *thread*. Threads are most familiar as a software model (see Chapter 10), but they are also a hardware model. In the usual hardware model, a thread has no explicit dependencies with instructions in any other thread, although there may be implicit dependencies through operations on the same memory address. The critical issues are (1) How many threads issue operations in each clock cycle? and (2) How many clock cycles does it take to switch between different threads?

Simultaneous multithreading (SMT) [947] allows many threads to issue instructions in each clock cycle. For example, if there are four threads and four functional units, then as long as each functional unit is needed by some thread in each clock cycle, all functional units can be kept busy every cycle, providing maximum use of the CPU hardware. The compiler or programmer must divide the program into separately executing threads. The SMT approach is starting to show up in CPU designs including versions of the IBM Power processors.

Fine-grained multithreading uses a single thread at a time but allows the CPU to change threads in a single clock cycle. Thus, a thread that must wait for a slow operation (anything from a floating-point addition to a load from main memory) can be "set aside," allowing other threads to run. Since a load from main memory may take 100 cycles or more, the benefit of this approach for hiding memory latency is apparent. The drawback when used to hide memory latency can be seen by applying Little's law. Large numbers of threads must be provided for this approach to succeed in completely hiding the latency of main (rather than cache) memory. The Cray MTA is the only commercial architecture to offer enough threads for this purpose.

All of these techniques can be combined. For example, fine-grained multithreading can be combined with superscalar ILP or explicit parallelism. SMT can restrict groups of threads to particular functional units in order to simplify the processor design, particular in processors with multiple FPUs and ALUs.

2.2.4 I/O and Networks

Just as in the uniprocessor case, I/O and networking for parallel processors have not received the same degree of attention as have CPU and memory performance. Fortunately, the lower performance levels of I/O and networking

devices relative to CPU and memory allow a simpler and less expensive architecture. On the other hand, lower performance puts tremendous strain on the architect trying to maintain balance in the system. A common I/O solution for parallel computers, particularly clusters, is not a parallel file system but rather a conventional file system, accessed by multiple processors.

Recall that data caches are often used to improve the performance of I/O systems in uniprocessors. As we have seen, it is important to maintain consistency between the different caches and between caches and memory if correct data is to be provided to programs. Unfortunately, particularly for networked file systems such as NFS, maintaining cache consistency seriously degrades performance. As a result, such file systems allow the system administrator to trade performance against cache coherence. For environments where most applications are not parallel and do not have multiple processes accessing the same file at the nearly the same time, cache coherence is usually sacrificed in the name of speed.

The redundant arrays of inexpensive Disks (RAID) approach is an example of the benefits of parallelism in I/O. RAID was first proposed in 1988 [757], with five different levels representing different uses of multiple disks to provide fault tolerance (disks, being mechanical, fail more often than entirely electronic components) and performance, while maintaining a balance between read rates, write rates, and efficient use of storage. The RAID approach has since been generalized to additional levels. Both hardware (RAID managed by hardware, presenting the appearance of a single but faster and/or more reliable disk) and software (separate disks managed by software) versions exist.

Parallel I/O can also be achieved by using arrays of disks arranged in patterns different from those described by the various RAID levels. Chapter 11 describes parallel I/O from the programmer's standpoint. A more detailed discussion of parallel I/O can be found in [658].

The simplest form of parallelism in networks is the use of multiple paths, each carrying part of the traffic. Networks within a computer system often achieve parallelism by simply using separate wires for each bit. Less tightly coupled systems, such as Beowulf clusters, sometimes use a technique called *channel bonding*, which uses multiple network paths, each carrying part of the message. GridFTP [22] is an example of software that exploits the ability of the Internet to route data over separate paths to avoid congestion in the network.

A more complex form of parallelism is the use of different electrical or optical frequencies to concurrently place several messages on the same wire or fiber. This approach is rarely used within a computer system because of the added cost and complexity, but it is used extensively in long-distance networks. New techniques for optical fibers, such as dense wavelength division multiplexing (DWDM), will allow a hundred or more signals to share the same optical fiber, greatly increasing bandwidth.

2.2.5 Support for Programming Models

Special operations are needed to allow processes and threads that share the same address space to coordinate their actions. For example, one thread may need to keep others from reading a location in memory until it has finished modifying that location. Such protection is often provided by *locks*: any thread that wants to access the particular data must first acquire the lock, releasing the lock when it is done. A lock, however, is not easy to implement with just load and store operations (though it can be done). Instead, many systems provide compound instructions that can be used to implement locks, such as test-and-set or fetch-and-increment. RISC systems often provide a "split" compound instruction that can be used to build up operations such as fetch-and-increment based on storing a result after reading from the same address only if no other thread or process has accessed the same location since the load.

Because rapid synchronization is necessary to support fine-grained parallelism, some systems (particularly PVPs) use special registers that all CPUs can access. Other systems have provided extremely fast *barriers*: no process can leave a barrier until all have entered the barrier. In a system with a fast barrier, a parallel system can be viewed as sequentially consistent, where an "operation" is defined as the group of instructions between two barriers. This provides an effective programming model for some applications.

In distributed-memory machines, processes share no data and typically communicate through messages. In shared-memory machines, processes directly access data. There is a middle ground: remote memory access (RMA). This is similar to the network-connected distributed-memory system except that additional hardware provides put and get operations to store to or load from memory in another node. The result is still a distributed-memory machine, but one with very fast data transfers. Examples are the Compaq AlphaServer SC, Cray T3D and T3E, NEC Cenju 4, and Hitachi SR8000.

2.2.6 Summary

Parallelism is a powerful approach to improving the performance of a computer system. All systems employ some degree of parallelism, even if it is only parallel data paths between the memory and the CPU. Parallelism is particularly good at solving problems related to bandwidth or throughput; it is less effective at dealing with latency or startup costs (although the ability to switch between tasks provides one way to hide latency as long as enough independent tasks can be found). Parallelism does not come free, however. The effects of memory latency are particularly painful, forcing complex consistency models on the programmer and difficult design constraints on the hardware designer.

In the continuing quest for ever greater performance, today's parallel computers often combine many of the approaches discussed here. One of the most popular is distributed-memory clusters of nodes, where each node is a shared-memory processor, typically with 2 to 16 processors, though some clusters have SMP nodes with as many as 128 processors. Another important class of machine is the parallel vector processor; this uses vector-style CPU parallelism combined

with shared memory.

We emphasize that hardware models and software (or programming) models are essentially disjoint; shared-memory hardware provides excellent message-passing support, and distributed-memory hardware can (at sometimes substantial cost) support a shared-memory programming model.

We close this section with a brief mention of taxonomies of parallel computers. A taxonomy of parallel computers provides a way to identify the important features of a system. Flynn [341] introduced the best-known taxonomy that defines four different types of computer based on whether there are multiple data streams and/or multiple instruction streams. A conventional uniprocessor has a single instruction stream and a single data stream and is denoted SISD. Most of the parallel computers that we have described in this section have both multiple data and multiple instruction streams (because they have many memories and CPUs); these are called MIMD. The single instruction but multiple data parallel computer, or SIMD, has already been mentioned. The fourth possibility is the multiple instruction, single data, or MISD; this category is not used. A standard taxonomy for MIMD architectures has not yet emerged, but it is likely to be based on whether the memory is shared or distributed and, if it is shared, whether it is cache coherent and how access time varies. Many of the terms used to describe these alternatives have been discussed above, including UMA, CC-NUMA, and DSM.

The term *single program, multiple data* (SPMD) is inspired by Flynn's taxonomy. Because the single program has branches and other control-flow constructs, SPMD is a subset of MIMD, not a subset of SIMD programs. Using a single program, however, does provide an important simplification for software, and most parallel programs in technical and scientific computing are SPMD.

2.3 Future Directions for Parallel Architectures

In some ways, the future of parallel architectures, at least for the next five years, is clear. Most parallel machines will be hybrids, combining nodes containing a modest number of commodity CPUs sharing memory in a distributed-memory system. Many users will have only one shared-memory node; for them, shared-memory programming models will be adequate. In the longer term, the picture is much hazier. Many challenges will be difficult to overcome. Principal among these are memory latency and the limits imposed by the speed of light. Heat dissipation is also becoming a major problem for commodity CPUs. One major contributor to the increase in clock speeds for CPUs has been a corresponding decrease in the size of the features on the CPU chip. These feature sizes are approaching the size of a single atom, beyond which no further decrease is possible.

While these challenges may seem daunting, they offer an important opportunity to computer architects and software scientists—an opportunity to take a step that is more than just evolutionary.

As we have discussed above, one of the major problems in designing any computer is providing a high-bandwidth, low-latency path between the CPU

and memory. Some of this cost comes from the way DRAMs operate: data is stored in rows; when an item is needed, the entire row is read and the particular bit is extracted; the other bits in the row are discarded. This simplifies the construction of the DRAM (separate wires are not needed to get to each bit), but it throws away significant bandwidth. Observing that DRAM densities are increasing at a rate even faster than the rate at which commodity software demands memory, several researchers have explored combining the CPU and memory on the same chip and using the entire DRAM row rather than a single bit at a time. In fact, an early commercial version of this approach, the Mitsubishi M32000D3 processor, used a conventional, cache-oriented RISC processor combined with memory and organized so that a row of the memory was a cache line, allowing for enormous (for the time) bandwidth in memory-cache transfers. Several different architectures that exploit processors and memory in the same chip are currently being explored [275, 756], including approaches that consider vector-like architectures and approaches that place multiple processors on the same chip. Other architects are looking at parallel systems built from such chips; the IBM Blue Gene [122] project expects to have a million processor system (with around 32 processors per node).

Superconducting elements promise clock speeds of 100 GHz or more. Of course, such advances will only exacerbate the problem of the mismatch between CPU and memory speeds. Designs for CPUs of this kind often rely on hardware multithreading techniques to reduce the impact of high memory latencies.

Computing based on biological elements often seeks to exploit parallelism by using molecules as processing elements. Quantum computing, particularly quantum computing based on exploiting the superposition principle, is a fundamentally different kind of parallelism.

2.4 Conclusion

Parallel architecture continues to be an active and exciting area of research. Most systems now have some parallelism, and the trends point to increasing amounts of parallelism at all levels, from 2 to 16 processors on the desktop to tens to hundreds of thousands for the highest-performance systems. Systems continue to be developed; see [955] for a review of current supercomputers, including large-scale parallel systems.

Access to memory continues to be a major issue; hiding memory latency is one area where parallelism doesn't provide a (relatively) simple solution. The architectural solutions to this problem have included deep memory hierarchies (allowing the use of low-latency memory close to the processor), vector operations (providing a simple and efficient "prefetch" approach), and fine-grained multithreading (enabling other work to continue while waiting for memory). In practice, none of these approaches completely eliminates the problem of memory latency. The use of low-latency memories, such as caches, suffers when the data does not fit in the cache. Vector operations require a significant amount of regularity in the operations that may not fit the best (often adaptive) algorithms, and multithreading relies on identifying enough independent threads. Because

of this, parallel programming models and algorithms have been developed that allow the computational scientist to make good use of parallel systems. That is the subject of the rest of this book.

Chapter 9

Software Technologies

Ian Foster, Jack Dongarra, Ken Kennedy, and Charles Koelbel

While parallel computing is defined by hardware technology, it is software that renders a parallel computer usable. Parallel software technologies are the focus of both this overview chapter and the seven more comprehensive chapters that follow in this part.

The concerns of the parallel programmer are those of any programmer: algorithm design, convenience of expression, efficiency of execution, ease of debugging, component reuse, and life-cycle issues. Hence, we should not be surprised to find that the software technologies required to support parallel program development are familiar in terms of their basic function. In particular, the parallel programmer, like any programmer, requires: languages and/or application programming interfaces (APIs) that allow for the succinct expression of complex algorithms, hiding unimportant details while providing control over performance-critical issues; associated tools (e.g., performance profilers) that allow diagnosis and correction of errors and performance problems; and convenient formulations of efficient algorithms for solving key problems, ideally packaged so that they can easily be integrated into an application program.

However, despite these commonalities, the particular characteristics of parallel computers and of parallel computing introduce additional concerns that tend to complicate both parallel programming and the development of parallel programming tools. In particular, we must be concerned with the following three challenges:

1. *Concurrency and communication.* Parallel programs may involve the creation, coordination, and management of potentially thousands of independent threads of control. Interactions between concurrent threads of control may result in nondeterminism. These issues introduce unique concerns that have profound implications for every aspect of the program

development process.

2. *Need for high performance.* In sequential programming, ease of expression may be as important or even more important than program performance. In contrast, the motivation for using parallel computation is almost always a desire for high performance. This requirement places stringent constraints on the programming models and tools that can reasonably be used for parallel programming.

3. *Diversity of architecture.* The considerable diversity seen in parallel computer architectures makes the development of standard tools and portable programs more difficult than is the case in sequential computing, where we find remarkable uniformity in basic architecture.

The role of parallel software is thus to satisfy the requirements listed at the beginning of this section, while simultaneously addressing in some fashion the three challenges of concurrency and communication, performance demands, and architectural diversity. This is a difficult task, and so in practice we find a variety of approaches to parallel software, each making different trade-offs between these requirements.

In the rest of this chapter, we provide an overview of the major software and algorithmic technologies that we can call upon when developing parallel programs. In so doing, we revisit issues that were first developed at the beginning of the book, relating to programming models, methodologies, and technologies, and attempt to integrate those different perspectives in a common framework. We structure the presentation in terms of two key questions that we believe will be asked by any parallel programmer:

- *How do I select the parallel programming technology (library or language) to use when writing a program?* We introduce the programming models, APIs, and languages that are commonly used for parallel program development and provide guidance concerning when these different models, APIs, and languages may be appropriate.

- *How do I achieve correct and efficient execution?* Here, we discuss issues relating to nondeterminism and performance modeling.

In each case, we provide pointers to the chapters in which these issues are discussed at greater length.

A third important question, *How do I reuse existing parallel algorithms and code?*, is addressed in Chapter 17, where we describe several techniques used to achieve code reuse in parallel algorithms.

9.1 Selecting a Parallel Program Technology

As was explained in Chapter 2, a parallel computer is a collection of processing and memory elements, plus a communication network used to route requests and information among these elements. The task of the parallel programmer is

to coordinate the operation of these diverse elements so as to achieve efficient and correct execution on the problem of interest.

The performance of a parallel program is determined by how effectively it maximizes *concurrency* (the number of operations that can be performed simultaneously) while minimizing the amount of *communication* required to access "nonlocal" data, transfer intermediate results, and synchronize the operation of different threads of control. Communication costs are frequently sensitive to *data distribution*, the mapping of application data structures to memory elements; a good data distribution can reduce the number of memory accesses that require expensive communication operations. If work is not distributed evenly among processors, *load imbalances* may occur, reducing concurrency and performance.

When evaluating the correctness of a parallel program, the programmer may need to take into account the possibility of *race conditions*, which occur when the executions of two or more distinct threads of control are sufficiently unconstrained that the result of a computation can vary nondeterministically, depending simply on the speed at which different threads proceed.

The programmer, when faced with the task of writing an efficient and correct parallel program, can call upon a variety of parallel languages, compilers, and libraries, each of which implements a distinct programming model with different trade-offs between ease of use, generality, and achievable performance.

In the rest of this section, we first review some of the principal programming models implemented by commonly used languages and libraries. Then, we examine each of these languages and libraries in turn and discuss their advantages and disadvantages.

9.1.1 Parallel Programming Models

We first make some general comments concerning the programming models that underlie the various languages and libraries that will be discussed subsequently.

Thirty years of research have led to the definition and exploration of a large number of parallel programming models [870]. Few of these models have survived, but much experience has been gained in what is useful in practical settings.

Data parallelism versus task parallelism. Parallel programs may be categorized according to whether they emphasize concurrent execution of the same task on different data elements (*data* parallelism) or the concurrent execution of different tasks on the same or different data (*task* parallelism). For example, a simulation of galaxy formation might require that essentially the same operation be performed on each of a large number of data items (stars); in this case, a data-parallel algorithm is obtained naturally by performing this operation on multiple items simultaneously. In contrast, in a simulation of a complex physical system comprising multiple processes (e.g., a multidisciplinary optimization of an aircraft might couple airflow, structures, and engine simulations), the different components can be executed concurrently, hence obtaining task parallelism.

Most programs for scalable parallel computers are data parallel in nature,

for the simple reason that the amount of concurrency that can be obtained from data parallelism tends to be larger than can be achieved via task parallelism. Nevertheless, task parallelism can have an important role to play as a software engineering technique: it often makes sense to execute distinct components on disjoint sets of processors (or even on different computers) for modularity reasons. It is increasingly common for parallel programs to be structured as a task-parallel composition of data-parallel components.

Explicit versus implicit parallelism. Parallel programming systems can be categorized according to whether they support an explicitly or implicitly parallel programming model. An *explicitly* parallel system requires that the programmer specify directly the activities of the multiple concurrent "threads of control" that form a parallel computation. In contrast, an *implicitly* parallel system allows the programmer to provide a higher-level specification of program behavior in which parallelism is not represented directly. It is then the responsibility of the compiler or library to implement this parallelism efficiently and correctly.

Implicitly parallel systems can simplify programming by eliminating the need for the programmer to coordinate the execution of multiple processes. For example, in the implicitly parallel, primarily data-parallel language High Performance Fortran, the programmer writes what is essentially sequential Fortran 90 code, augmented with some directives. Race conditions cannot occur, and the HPF program need not be rewritten to take advantage of different parallel architectures.

Explicitly parallel systems provide the programmer with more control over program behavior and hence can often be used to achieve higher performance. For example, an MPI implementation of an adaptive mesh-refinement algorithm may incorporate sophisticated techniques for computing mesh distributions, for structuring communications among subdomains, and for redistributing data when load imbalances occur. These strategies are beyond the capabilities of today's HPF compilers.

A parallel programming style that is becoming increasingly popular is to encapsulate the complexities of parallel algorithm design within libraries (e.g., an adaptive-mesh-refinement library, as just discussed). An application program can then consist of just a sequence of calls to such library functions. In this way, many of the advantages of an implicitly parallel approach can be obtained within an explicitly parallel framework.

Shared memory versus distributed memory. Explicitly parallel programming systems can be categorized according to whether they support a shared- or distributed-memory programming model. In a *shared-memory* model, the programmer's task is to specify the activities of a set of processes that communicate by reading and writing shared memory. In a *distributed-memory* model, processes have only local memory and must use some other mechanism (e.g., message passing or remote procedure call) to exchange information.

Shared-memory models have the significant advantage that the programmer need not be concerned with data-distribution issues. On the other hand, high-

performance implementations may be difficult on computers that lack hardware support for shared memory, and race conditions tend to arise more easily.

Distributed-memory models have the advantage that programmers have explicit control over data distribution and communication; this control facilitates high-performance programming on large distributed-memory parallel computers.

Other programming paradigms. Some important aspects of programming are orthogonal to the model of parallelism, but can have a significant impact on the parallel programming process. Arguably the most important example of this is *object-oriented programming*. Although the fundamentals of object-oriented design — including encapsulation of data and function, inheritance and polymorphism, and generic programming enabled by powerful abstraction — come from the sequential world, they are also relevant to parallel computing. In particular, appropriate abstractions can hide parallelism when it complicates programming. Good examples include C++ libraries for array expressions [800, 963], which provide a familiar interface to the programmer that can be used efficiently on both sequential and parallel computers. Abstractions can also provide "glue" to tie parallel components together. The resulting component architectures provide excellent runtime environments for building applications by composition. Chapter 13 explores object-oriented parallel programming in more detail.

Other programming paradigms are specific to parallel programming, but not to the particular model of parallelism. One of particular interest is the *single program, multiple data* approach, in which all available processors execute the same textual program. Because each processor has its own thread of control, different processors may take different paths through this program and may operate on different data. (In fact, it is usually the case that most processors take similar paths through the program, perhaps varying slightly in the number of iterations of a particular loop or the active branch of some conditionals.) Many programmers find this an intuitive model that can be used for shared or distributed memory. For example, they often write message-passing programs in this style, although it is not mandated by the MPI standard. Some explicit parallel languages make heavy use of the SPMD paradigm, as Section 12.4 describes.

9.1.2 Parallel Programming Technologies

We provide a brief summary of the major programming technologies discussed in this book and provide pointers to the chapters where they are covered in more detail. In the next subsection, we discuss the situations in which each is to be preferred.

Message-Passing Interface

The Message-Passing Interface (MPI) is a specification for a set of functions for managing the movement of data among sets of communicating processes. Official MPI bindings are defined for C, Fortran, and C++; bindings for var-

ious other languages have been produced as well. MPI defines functions for point-to-point communication between two processes, for collective operations among processes, for parallel I/O, and for process management. In addition, MPI's support for *communicators* facilitates the creation of modular programs and reusable libraries. Communication in MPI specifies the types and layout of data being communicated, allowing MPI implementations to both optimize for noncontiguous data in memory and support clusters of heterogeneous systems. As illustrated in Figure 9.1, taken from Chapter 16, MPI programs are commonly implemented in terms of a Single Program Multiple Data (SPMD) model, in which all processes execute essentially the same logic. Chapter 10 provides more details on MPI, while MPI's support for parallel I/O is discussed in Chapter 11.

Analysis. MPI is today the technology of choice for constructing *scalable* parallel programs, and its ubiquity means that no other technology can beat it for portability. In addition, a significant body of MPI-based libraries has emerged that provide high-performance implementations of commonly used algorithms. Nevertheless, given that programming in an explicit message-passing style can place an additional burden on the developer, other technologies can be useful if our goal is a modestly parallel version of an existing program (in which case OpenMP may be useful), we are using Fortran 90 (HPF), or our application is a task-parallel composition designed to execute in a distributed environment(CORBA, RMI).

Parallel Virtual Machine

Parallel Virtual Machine (PVM) represents another popular instantiation of the message-passing model that was one of the principal forerunners of MPI and the first de facto standard for implementation of portable message-passing programs. The example in Figure 9.2 shows a PVM version of the Poisson problem that was previously given in MPI. Although PVM has been superseded by MPI for tightly coupled multiprocessors, it is still widely used on networks of workstations. PVM's principal design goal was portability, even to nonhomogeneous collections of nodes, which was gained by sacrificing optimal performance. MPI, on the other hand, provides high-performance communication. MPI-1 provided only a nonflexible static process model, while MPI-2 adds a scalable dynamic process model.

Central to the design of PVM is the notion of a "virtual machine" — a set of heterogeneous hosts connected by a network that appears logically to the user as a single large parallel computer. PVM API functions provide the ability to (1) join or leave the virtual machine; (2) start new processes by using a number of different selection criteria, including external schedulers and resource managers; (3) kill a process; (4) send a signal to a process; (5) test to check that a process is responding; and (6) notify an arbitrary process if another disconnects from the PVM system.

Analysis. If an application is going to be developed and executed on a single MPP, then MPI has the advantage of expected higher communication perfor-

```
use mpi
real u(0:n,js-1:je+1), unew(0:n,js-1:je+1)
real f(1:n-1, js:je), h
integer nbr_down, nbr_up, status(MPI_STATUS_SIZE), ierr

! Code to initialize f, u(0,*), u(n:*), u(*,0), and
! u(*,n) with g

h = 1.0 / n
do k=1, maxiter
  ! Send down
  call MPI_Sendrecv( u(1,js), n-1, MPI_REAL, nbr_down, k, &
                     u(1,je+1), n-1, MPI_REAL, nbr_up, k, &
                     MPI_COMM_WORLD, status, ierr )
  ! Send up
  call MPI_Sendrecv( u(1,je), n-1, MPI_REAL, nbr_up, k+1, &
                     u(1,js-1), n-1, MPI_REAL, nbr_down, k+1,&
                     MPI_COMM_WORLD, status, ierr )
  do j=js, je
    do i=1, n-1
      unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                           u(i,j+1) + u(i,j-1) - &
                           h * h * f(i,j) )
    enddo
  enddo
  ! code to check for convergence of unew to u.
  ! Make the new value the old value for the next iteration
  u = unew
enddo
```

Figure 9.1: Message-passing version of the Poisson problem, as presented in Figure 16.3.

mance. In addition, MPI has a much richer set of communication functions, so it is favored when an application is structured to exploit special communication modes, such as nonblocking send, not available in PVM. PVM has the advantage when the application is going to run over a networked collection of hosts, particularly if the hosts are heterogeneous. PVM includes resource management and process control functions that are important for creating portable applications that run on clusters of workstations and MPPs. PVM is also to be favored when fault tolerance is required. MPI implementations are improving in all of these areas, but PVM still provides better functionality in some settings.

Parallelizing Compilers

Since parallel programming is so hard, it is appealing to think that the best solution would be to let the compiler do it all. Thus, automatic parallelization — extraction of parallelism from sequential code by the compiler — has been

```
real u(0:n,js-1:je+1), unew(0:n,js-1:je+1)
real f(1:n-1, js:je), h
integer nbr_down, nbr_up, rc

! Code to initialize f, u(0,*), u(n:*), u(*,0), and
! u(*,n) with g

h = 1.0 / n
do k=1, maxiter
  ! Send down
  ! Can not do a head to head as lack of buffering will cause
  ! deadlock, so odd rows sends first
  ! while even receives and then we swap
  if (odd) then
     call pvmfpsend(nbr_down, k, u(1,js),   n-1, PVM_FLOAT, rc)
     call pvmfprecv(nbr_up,   k, u(1,je+1), n-1, PVM_FLOAT, rc)
  else
     call pvmfprecv(nbr_up,   k, u(1,je+1), n-1, PVM_FLOAT, rc)
     call pvmfpsend(nbr_down, k, u(1,js),   n-1, PVM_FLOAT, rc)
  endif

  ! Send up
  ! Similar odd/even swapping to sending down
  if (odd) then
     call pvmfpsend(nbr_up,   k, u(1,je),   n-1, PVM_FLOAT, rc)
     call pvmfprecv(nbr_down, k, u(1,js-1), n-1, PVM_FLOAT, rc)
  else
     call pvmfprecv(nbr_down, k, u(1,js-1), n-1, PVM_FLOAT, rc)
     call pvmfpsend(nbr_up,   k, u(1,je),   n-1, PVM_FLOAT, rc)
  endif

  do j=js, je
    do i=1, n-1
      unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                           u(i,j+1) + u(i,j-1) - &
                           h * h * f(i,j) )
    enddo
  enddo
  ! code to check for convergence of unew to u.
  ! Make the new value the old value for the next iteration
  u = unew
enddo
```

Figure 9.2: A PVM formulation of the Poisson problem.

the holy grail of parallel computing software, particularly given the success of automatic methods on vector machines. Unfortunately, automatic parallelization has never achieved success comparable to that of automatic vectorization. As explained in Chapter 12, the reasons for this failure stem from the greater complexity of compiler analysis and hardware features of parallel machines.

As a result of these difficulties, automatic parallelization has been successful primarily on shared-memory machines with small numbers of processors. The performance gains that can be expected from this technology are application dependent, but are generally small. Programmer-supplied information (typically communicated via directives) can improve the overall parallelization in some situations. However, in those cases OpenMP offers a much more portable way of specifying that information.

Analysis. Parallelizing compilers are certainly worth trying, especially when first implementing a program on a shared-memory machine where only a small degree of parallelism is required. If more parallelism or portability is needed, OpenMP or MPI are better solutions. On scalable machines, HPF may provide an acceptably simple alternative to automatic parallelization for regular, data-parallel problems coded in Fortran 90.

P-threads

As noted above, in the shared-memory programming model, multiple threads of control operate in a single memory space. The POSIX standard threads package (P-threads) represents a particularly low-level, but widely available, implementation of this model. The P-threads library provides functions for creating and destroying threads and for coordinating thread activities via constructs designed to ensure exclusive access to selected memory locations (locks and condition variables). Chapter 10 provides a more detailed discussion of P-threads.

Analysis. We do *not* recommend the use of P-threads as a general-purpose parallel program development technology. While they have their place in specialized situations and in the hands of expert programmers, the unstructured nature of P-threads constructs makes the development of correct and maintainable programs difficult. In addition, P-threads programs are not scalable to large numbers of processors.

OpenMP

An alternative approach to shared-memory programming is to use more structured constructs such as parallel loops to represent opportunities for parallel execution. This approach is taken in the increasingly popular OpenMP, a set of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran and C/C++ programs. As illustrated in Figure 9.3, OpenMP extensions focus on the exploitation of parallelism within loops. In the example, the outer loop of a finite-difference calculation is declared to be parallel. Arrays `u`, `unew` and `f` are shared, while the inner loop induction variable is private. OpenMP also provides the `WORKSHARE`

```
real u(0:n,0:n), unew(0:n,0:n)
real f(1:n-1, 1:n-1), h

! Code to initialize f, u(0,*), u(n:*), u(*,0), and
! u(*,n) with g

h = 1.0 / n
do k=1, maxiter

  !$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i)
  do j=1, n-1
    do i=1, n-1
      unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                           u(i,j+1) + u(i,j-1) - &
                           h * h * f(i,j) )
    enddo
  enddo
  !$OMP END PARALLEL DO

  ! code to check for convergence of unew to u.

  ! Make the new value the old value for the next iteration
  !$OMP PARALLEL WORKSHARE
  u = unew
  !$OMP END PARALLEL WORKSHARE

enddo
```

Figure 9.3: An OpenMP formulation of the Poisson problem.

directive to exploit the data parallelism in array assignments (and other situations not shown here).

The parallelism in OpenMP may be coarse grained (as in the `do` loop of Figure 9.3) or fine grained (as in the array assignment, or as it would be if nested parallelism were used to make every iteration of both loops parallel). A desirable feature of OpenMP is that it preserves sequential semantics: in a legal program, you may ignore the structured comments and a sequential program is obtained. This simplifies program development, debugging, and maintenance. However, if the programmer makes an error and mistakenly specifies a loop to be parallel when a race condition exists, the program will produced undetermined results. Even though such a program is "noncompliant," it may be difficult to find the problem. Chapter 12 provides a more detailed discussion of OpenMP.

Analysis. We recommend the use of OpenMP when the goal is to achieve modest parallelism on a shared-memory computer. In this environment, the simplicity of the OpenMP model and the fact that a parallel program can be obtained via the incremental addition of directives to a sequential program are significant advantages. On the other hand, the lack of support for user manage-

```
real u(0:n,0:n), unew(0:n,0:n)
real f(1:n-1, 1:n-1), h
!HPF$ PROCESSORS pr(4)            ! Run on 4 processors
!HPF$ DISTRIBUTE u(BLOCK,*)       ! Distribute u by rows
!HPF$ ALIGN unew(i,j) WITH u(i,j) ! Distribute unew like u
!HPF$ ALIGN f(i,j) WITH u(i,j)    ! Distribute f like u

! Code to initialize f, u(0,*), u(n:*), u(*,0), and
! u(*,n) with g

h = 1.0 / n
do k=1, maxiter

  unew(1:n-1,1:n-1) =  0.25 * ( u(2:n,1:n-1) + u(0:n-2,1:n-1) + &
                           u(1:n-1,2:n) + u(1:n-1,0:n-2) - &
                           h * h * f(1:n-1,1:n-1) )

  ! code to check for convergence of unew to u.

  ! Make the new value the old value for the next iteration
  u = unew

enddo
```

Figure 9.4: An HPF formulation of the Poisson problem.

ment of data distribution means that scalable implementations of OpenMP are unlikely to appear in the foreseeable future.

High Performance Fortran

High Performance Fortran (HPF), like OpenMP, extends a sequential base language (in this case Fortran 90) with a combination of directives, library functions, and (in the case of HPF) some new language constructs to provide a data-parallel, implicitly parallel programming model. HPF differs from OpenMP in its focus on support for user management of data distribution, so as to support portable, high-performance execution on scalable computers of all kinds, particularly in distributed-memory environments.

Figure 9.4 illustrates how structured comments are used to express the number of processors that a program is to run on and to control the distribution of data. Notice that only three directives have been added to what is otherwise a pure Fortran 90 program: `PROCESSORS`, `DISTRIBUTE`, and `ALIGN` directives. These directives partition each of the arrays by contiguous blocks of rows, hence allocating approximately $\frac{n}{4}$ rows to each of 4 processors. Although the array assignment statement is explicitly parallel, the same distribution applied to a loop formulation of finite difference would be expected to achieve the same results — if the loop can be run in parallel on the assigned collection of processors, it

```
real u(0:n,js-1:je+1)[*], unew(0:n,js-1:je+1)[*]
real f(1:n-1, js:je), h
integer nbr_down, nbr_up, nbrs(3), me

! Code to initialize f, u(0,*), u(n:*), u(*,0), and
! u(*,n) with g

h = 1.0 / n
do k=1, maxiter
  ! Send down
  u(1:n-1,je+1)[nbr_down] = u(1:n-1,js)[me]
  ! Send up
  u(1:n-1,js-1)[nbr_up] = u(1:n-1,je)[me]
  call synch_all( wait=nbrs )
  do j=js, je
    do i=1, n-1
      unew(i,j)[me] = 0.25 * ( u(i+1,j)[me] + u(i-1,j)[me] + &
                           u(i,j+1)[me] + u(i,j-1)[me] - &
                           h * h * f(i,j) )
    enddo
  enddo
  ! code to check for convergence of unew to u.
  ! Make the new value the old value for the next iteration
  u[me] = unew[me]
enddo
```

Figure 9.5: A Co-Array Fortran formulation of the Poisson problem.

should be. Chapter 12 provides more details on HPF.

Analysis. When HPF works well, it is a wonderful tool: complex parallel algorithms can be expressed succinctly as Fortran 90 code. Furthermore, it captures, in a machine-independent way, the notion of data decomposition, which is essential to successful parallelization of codes for distributed-memory systems. However, current implementations of HPF are effective primarily for algorithms defined on regular grids and for dense linear algebra. Although the extended HPF-2 standard defines language extensions that would make HPF applicable to irregular computations, few compilers implement these extensions. Hence, even though there are a number of substantive applications, HPF remains a niche technology, for now at least.

Co-Array Fortran

Co-Array Fortran [719] takes the approach of designing a language to express the single program multiple data programming model. Many threads (which may correspond to physical processors) execute the same program, exchanging data by assignment statements. Unlike MPI and PVM, only one thread in the data exchange needs to specify this communication. A program accomplishes

this by declaring nonlocally-accessed arrays with an extra "co-dimension" and indexing that co-dimension with the thread id. Figure 9.5 shows how this works in the Poisson example program. Note the explicit synchronization (the call to `synch_all`) that ensures that both data copies are completed. Chapter 12 discusses Co-Array fortran in more detail.

Analysis. SPMD languages like Co-Array Fortran have many potential advantages. They are often reasonably clean (as in the above example), allow low-level manipulations when necessary to tune program performance, and are not tied to particular hardware models. However, they are relatively new and not widely supported. In particular, Co-Array Fortran is not a formal standard, and is not supported on many machines. (The Cray C90 and T3E are important machines where it is supported.) Some research implementations are beginning to appear. While we hesitate to recommend such languages for production use today, the situation may improve in the future. Co-Array Fortran may also be a good choice for experimental, proof-of-concept work due to its expressibility.

POOMA and HPC++

An alternative approach to the implementation of implicit data parallelism is to define libraries that use object-oriented techniques (in particular, inheritance and polymorphism) to abstract and encapsulate parallel operations. This is essentially the approach taken in POOMA [800] and HPC++, two libraries that define standard-use object-oriented technology to define classes that encapsulate parallelism. In POOMA, for example, we can write code such as the following:

```
A[I][J] = 0.25*(A[I+1][J] + A[I-1][J] + A[I][J+1] + A[I][J-1] );
```

to express the data-parallel operation for which we have presented various implementations in this chapter. In this formulation, `A` is a two-dimensional field, and `I` and `J` are index objects representing the domain of the field object. As is discussed in more detail in Chapter 13, issues relating to distribution across processors and communication are handled by the array objects. A more complete version of this example is presented as Figure 9.6, which is replicated from Chapter 13. That chapter also explains how Java can be used as an effective programming language for object-oriented libraries.

Analysis. A significant advantage of object-oriented approaches is the great simplicity and clarity that can be obtained. Another advantage is that the developer of these libraries can incorporate substantial "smarts" in order to obtain good performance on parallel platforms (see Chapter 13). A disadvantage in some situations is that because we are dealing with often complex library software, the task of debugging performance and correctness problems can be nontrivial.

Component Models

The final programming technology that we mention briefly is the various component technologies that have been proposed and are used to facilitate the modular construction of complex software systems. CORBA, .COM, and Java Beans are

```
01  #include "Pooma/Arrays.h"
02
03  #include <iostream>
04
05  // The size of each side of the domain.
06  const int N = 20;
07
08  int
09  main(
10      int                 argc,           // argument count
11      char*               argv[]          // argument list
12  ){
13      // Initialize POOMA.
14      Pooma::initialize(argc, argv);
15
16      // The array we'll be solving for
17      Array<2,double> V(N, N);
18      V = 0.0;
19
20      // The right hand side of the equation (spike in the center)
21      Array<2,double> b(N, N);
22      b = 0.0;
23      b(N/2, N/2) = -1.0;
24
25      // Specify the interior of the domain
26      Interval<1> I(1, N-2), J(1, N-2);
27
28      // Iterate 200 times
29      for (int iteration=0; iteration<200; ++iteration)
30      {
31          V(I,J) = 0.25*(V(I+1,J) + V(I-1,J) + V(I,J+1) + V(I,J-1) - b(I,J));
32      }
33
34      // Print out the result
35      std::cout << V << std::endl;
36
37      // Clean up POOMA and report success.
38      Pooma::finalize();
39      return 0;
40  }
```

Figure 9.6: A POOMA code that performs a Jacobi iteration with a 5-point stencil.

well-known examples. While various groups have experimented with the use of these technologies within high-performance computing (e.g., [553]), lack of support for parallelism has hindered their use for parallel computing proper. The Common Component Architecture effort [47], discussed in Chapter 13, is attempting to overcome some of these problems. Other relevant efforts include Problem Solving Environments (PSEs) and PSE Toolkits like Uintah [250], NetSolve [173], and WebFlow [356], discussed more fully in Chapter 14. We can hope that these efforts will produce the technology base required to support truly modular construction of parallel software systems. In the meantime, the parallel programmer can and should seek to apply well-established principles of modular design.

Hybrids

A variety of hybrid approaches are possible and in some cases are proving effective and popular. For example, it is increasingly common to see applications developed as a distributed-memory (MPI) framework with shared-memory parallelism (e.g., OpenMP) used within each "process." The primary motivation is a desire to write programs whose structure mirrors that of contemporary parallel computers consisting of multiple shared-memory computers connected via a network. The technique can have advantages: for example, a multidimensional problem can be decomposed across processes in one dimension and within a process in a second.

Other hybrids that have been discussed in a research context include MPI and P-Threads, MPI and HPF [350, 351], and CORBA and HPF.

9.1.3 Summary

In the preceding discussion of parallel programming models and technologies, we have made a number of points concerning the pros and cons of different approaches. Table 9.1 brings these various issues together in the form of a set of rules for selecting parallel programming models. We emphasize that this table includes only some of the available parallel technologies.

9.2 Achieving Correct and Efficient Execution

The problem of achieving *correct* and *efficient* parallel programs is made difficult by the issues noted in the introduction to this chapter: nondeterminism, concurrency, and complex parallel computer architectures. These problems can be overcome by a combination of good programming practice and appropriate tools. Tools such as debuggers, profilers, and performance analyzers are discussed in Chapter 15; we talk here about two issues of programming practice, namely dealing with nondeterminism and performance modeling.

9.2.1 Dealing with Nondeterminism

A nondeterministic computation is one in which the result computed depends on the order in which two or more unsynchronized threads of control happen to

Table 9.1: Decision rules for selecting parallel programming technologies.

Use ...	If:
Compilers	goal is to extract moderate [$\mathcal{O}$(4–10)] parallelism from existing code target platform has a good parallelizing compiler portability is not a major concern
OpenMP	goal is to extract moderate [$\mathcal{O}$(10)] parallelism from existing code good quality implementation exists for target platform portability to distributed-memory platforms is not a major concern
MPI	scalability is important application must run on some message-passing platforms portability is important a substantive coding effort is acceptable to achieve other goals
HPF	application is regular and data parallel a simple coding style in Fortran 90 is desirable explicit data distribution is essential to performance a high degree of control over parallelism is not critical
Co-Array Fortran	an implementation is available moderate coding effort (less than MPI) is desired SPMD programming style is acceptable
Threads	scalability is not important program involves fine-grained operations on shared data program has significant load imbalances OpenMP is not available or suitable
CORBA, RMI	program has task-parallel formulation interested in running in network-based system performance is not critical
High-level libraries	they address your specific problem the library is available on the target platform

execute. Nondeterministic interactions can sometimes be desirable: for example, they can allow us to select the "first" solution computed by a set of worker processes that are executing subtasks of unknown size. However, the presence of nondeterminism also greatly complicates the task of verifying program correctness; in principle, we need to trace every possible program execution before we can ensure that the program is correct. And in practice it can be difficult both to enumerate the set of possible executions and to reproduce a particular behavior. Hence, nondeterminism is to be avoided whenever possible. The following general techniques can be used to achieve this goal:

- When possible, use a parallel programming technology that does not permit race conditions to occur: e.g., HPF.

- If using a parallel programming technology that permits race conditions, adopt defensive programming practices to avoid unwanted nondeterminism. For example, in MPI, ensure that every "receive" call can match exactly one "send." Avoid the use of P-threads.
- When nondeterminism is required, encapsulate it within objects with well-defined semantics. For example, in a manager–worker structure, the manager may invoke a function "get next solution"; all nondeterminism is then encapsulated within this function.

9.2.2 Performance Modeling

In Chapter 15, tools are described for measuring and analyzing the performance of a parallel program. In principle, a good performance tool should be able to relate observed performance to the constructs of whatever parallel programming technology was used to write the original program. It may also seek to suggest changes to the program that can improve performance. Tools available today do not typically achieve this ideal, but they can provide useful information.

An important adjunct to any performance tool is the use of *analytic performance models* as a means of predicting likely performance and of explaining observed performance. As discussed for example in *Designing and Building Parallel Programs* [342], a good performance model relates parallel program performance (e.g., execution time) to key properties of the program and its target execution environment: for example, problem size, processor speed, and communication costs. Such a model can then be used for qualitative analysis of scalability. If the model is sufficiently accurate (and especially if it is calibrated with experimental data) it can also be used to explain observed performance. Performance models are also discussed in Chapter 15.

9.3 Future Directions

We conclude this chapter with a discussion of four areas in which significant progress is required — and, we believe, will occur — in parallel software concepts and technologies.

Clusters and DSM While shared-memory multiprocessors are becoming increasingly common, another parallel computing technology is also seeing widespread use, namely clusters constructed from PC nodes connected with commodity networks. Such clusters can be extremely cheap when compared with multiprocessors, but do not offer the same integrated operating system services or convenient shared-memory programming model. Heterogeneity is another potential obstacle. However, numerous research and development activities are working to overcome these problems.

At the operating system level, we see numerous activities focused on parallel file systems, scheduling, error management, and so on. In addition, work such as Fast Messages [734] and Virtual Interface Architecture (VIA) [966] is helping to reduce communication costs to something more like MPPs.

Clusters today are almost invariably programmed with MPI. Yet experience with multiprocessors shows that shared-memory parallelism can be more convenient for applications that involve irregular data structures and data access patterns. Hence, various groups are working to develop software-based distributed-shared-memory (DSM) systems that will allow a cluster to be treated as a shared-memory multiprocessor, with various combinations of runtime support, operating system modifications, and compiler modifications being used to provide (sometimes) efficient support for a shared-memory programming model.

Grids Emerging "Computational Grid" infrastructures support the coordinated use of network-connected computers, storage systems, and other resources, allowing them to be used as an integrated computational resource [347].

Grid concepts and technologies have significant implications for the practice of parallel computing. For example, while parallel computers have been used traditionally as "batch" engines for long-running, non-interactive jobs, in Grid environments a parallel computer may need to interact frequently with other systems, whether to acquire instrument data, enable interactive control, or access remote storage systems [345]. These new modes of use are likely to require new runtime system and resource management techniques [346].

Grid infrastructures can also be used to create what might be termed "generalized clusters," enabling the dynamic discovery and assembly of collections of resources that can be used to solve a computational problem. Because so many computational resources are underutilized, this mode of use has the potential to deliver order-of-magnitude increases in available computation. However, the heterogeneous and dynamic nature of such generalized clusters introduces significant challenges for algorithms and software technologies.

Ultra-scale computers The final architecture-based topic that we discuss relates to the software technologies required for tomorrow's extremely large-scale parallel computers — those capable of 10^{15} operations per second or more.

A variety of very different architectures have been proposed for such computers, ranging from scaled-up versions of today's commodity-based systems to systems based on processor-in-memory components and/or superconducting logic [900]. These different systems have in common a need to be able to exploit large amounts of parallelism — 10^3 times more than today's largest computers — and to deal with deep memory hierarchies in which memory may be a factor of 10^3 further away (in terms of processor clock cycles) than in today's systems.

These scaling issues, which derive from trends in processor and memory technology, pose major challenges for parallel software technologies at every level.

Programming Productivity One major goal for research and development in parallel computing must necessarily be to reduce the cost of writing and executing parallel programs, particularly for shared-memory multiprocessor systems. This goal becomes more difficult to achieve as the computing platforms become ever more complex. From the previous paragraphs, we can see that platforms will

become even more complex in the future. With these architectural advances, we may see the day that programming for the most advanced computational facilities will become the exclusive domain of professional programmers. This would be a major setback for computational science. To avoid that setback, we need revolutionary advances in programming support technologies.

As you will see in Chapter 12, automatic methods for extracting parallelism from conventional programming languages has been only a limited success. Furthermore, it seems unlikely that these techniques will extend well to the complex architectures of the future. So how can we support end-user programming while maintaining a high level of performance? One approach that shows considerable promise for productivity improvement is the use of high-level, domain-specific problem-solving environments. Examples of such systems include MATLAB [429], Mathematica [1005], Ellpack [494], and POOMA [53].

The difficulty with problem-solving environments as they are currently implemented is that they produce code that is not efficient enough to be used on a computation-intensive application. However, advanced techniques based on extensive library precompilation may offer a way to bring the performance of problem-solving languages up to the level of conventional languages [562]. Furthermore, by exploiting the domain knowledge contained in the language and the underlying library, it should be possible to extract the natural parallelism in the problem and tailor it to a variety of different target platforms.

Further Reading

An article by Skillicorn and Talia [870] provides an excellent survey of parallel programming paradigms and languages.

The book *Designing and Building Parallel Programs* [342] provides a good tutorial introduction to parallel computing, MPI, and HPF.

The book *The Grid: Blueprint for a Future Computing Infrastructure* [347] provides a comprehensive review of the technologies that underlie emerging Grid infrastructures and applications. See also [349] and [343].

Chapter 10

Message Passing and Threads

Ian Foster, William Gropp, and Carl Kesselman

In this chapter we examine two fundamental, although low-level, approaches to expressing parallelism in programs. Over the years, numerous different approaches to designing and implementing parallel programs have been developed (e.g., see the excellent survey article by Skillicorn and Talia [870]). However, over time, two dominant alternatives have emerged: *message passing* and *multithreading*.

These two approaches can be distinguished in terms of how concurrently executing segments of an application share data and synchronize their execution. In message passing, data is shared by explicitly copying ("sending") it from one parallel component to another, while synchronization is implicit with the completion of the copy. In contrast, the multithreading approach shares data implicitly through the use of shared memory, with synchronization being performed explicitly via mechanisms such as locks, semaphores and condition variables.

As with any set of alternatives, there are advantages and disadvantages to each approach. Multithreaded programs can be executed particularly efficiently on computers that use physically shared memory as their communication architecture. However, many parallel computers being built today do not support shared memory across the whole computer, in which case the message-passing approach is more appropriate.

From the perspective of programming complexity, the implicit sharing provided by the shared-memory model simplifies the process of converting existing sequential code to run on a parallel computer. However, the need for explicit synchronization can result in errors that produce nondeterministic race conditions that are hard to detect and correct. On the other hand, converting a program to use message passing requires more work up front, as one must ex-