CHAPTER 2

Parallel Computers

in which we briefly recount the history of parallel computers; we examine the characteristics of modern parallel computer hardware that influence parallel program design; and we introduce modern standard software libraries for parallel programming

CHAPTER 2 Parallel Computers

2.1 A Brief History of Parallel Computers

To understand how modern parallel computers are built and programmed, we must take a quick look at the history of parallel computers.

Up until the mid-1990s, there were no widely adopted standards in either parallel computer hardware or software. Many vendors sold parallel computers, but each vendor had its own proprietary designs for CPUs and CPU interconnection networks. Along with its proprietary hardware, each vendor provided its own proprietary languages and tools for writing parallel programs on its hardware—sometimes a variant of a scientific programming language such as Fortran, sometimes another language.

Because the hardware and software were mostly vendor-specific, parallel programs of that era tended not to be portable. You settled on a hardware vendor, then you used the vendor's supported parallel programming language to write your parallel programs. If you wanted to change vendors, you were faced with the dismaying prospect of rewriting and re-debugging all your programs using the new vendor's programming language.

As the twentieth century drew toward its close, four events took place that signaled the beginning of a paradigm shift for parallel computing. First, in the late 1970s, Robert Metcalfe and David Boggs invented the Ethernet local area network at the Xerox Palo Alto Research Center; in 1980, the 10-Mbps Ethernet standard was published by Digital Equipment Corporation, Intel, and Xerox; in 1983, a variation was standardized as IEEE Std 802.3. Second, in 1981, the Internet Protocol (IP) and the Transmission Control Protocol (TCP), developed by Vinton Cerf and Robert Kahn, were published as Internet standards—"Request For Comments" (RFC) 791 and RFC 793. Third, also in 1981, IBM started selling the IBM PC, whose open architecture became the de facto standard for personal computers. By 1983, for the first time in the history of computing, there was an open standard computer hardware platform (the PC), an open standard local area network (Ethernet) for interconnecting computers, and an open standard network protocol stack (TCP/IP) for exchanging data between computers. Fourth, in 1991, Linus Torvalds released the first version of the Linux operating system, a free, Unix-like operating system for the PC, that included an implementation of TCP/IP.

In 1995, Thomas Sterling, Donald Becker, and John Dorband at the NASA Goddard Space Flight Center, Daniel Savarese at the University of Maryland, and Udaya Ranawake and Charles Packer at the Hughes STX Corporation published a paper titled "Beowulf: a parallel workstation for scientific computation." In this paper, they described what is now called a cluster parallel computer, built from off-the-shelf PC boards, interconnected by an off-the-shelf Ethernet, using the Internet standard protocols for communication, and running the Linux operating system on each PC. The Beowulf cluster's performance was equal to that of existing proprietary parallel computers costing millions of dollars, but because it was built from off-the-shelf components, the Beowulf cluster cost only a fraction as much.

A year later, William Hargrove and Forrest Hoffman at Oak Ridge National Laboratory proved that a parallel computer could be built for zero dollars. Lacking a budget to buy a new parallel computer, they instead took obsolete PCs that were destined for the landfill, loaded them with the free Linux operating system, and hooked them up into a Beowulf cluster. Dubbing their creation the "Stone SouperComputer" (Figure 2.1), after the fable of the soldier who cooked up "stone soup" from a small stone along with a bit of this and a bit of that contributed by the villagers in the story, eventually Hargrove and Hoffman had a cluster with 191 nodes.

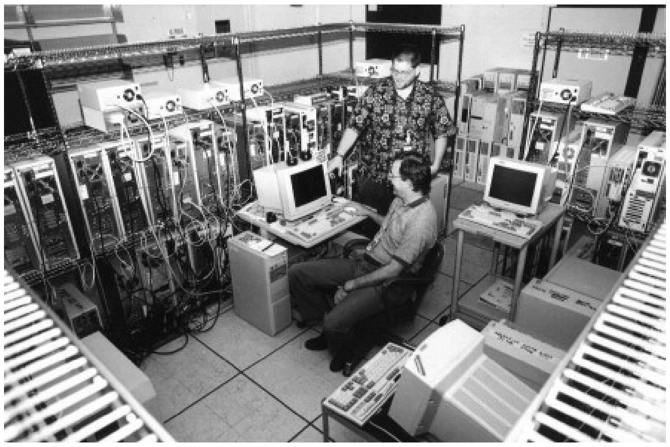
http://www.extremelinux.info/stonesoup/photos/1999-05/image01.jpg

Figure 2.1 The Stone Souper Computer

Once the PC, Ethernet, and TCP/IP became standardized, prices were driven down by mass production of microprocessor, memory, and Ethernet chips and cutthroat competition in the PC market. Today, computers are commodities you can buy in a store just like you can buy toasters and televisions—a state of affairs undreamed of when proprietary designs held sway. To get a parallel computer with a given amount of computing power, it usually costs much less to buy a multicore PC server, or to buy a bunch of PCs and a 1-Gbps Ethernet switch, from the corner computer store than to buy a proprietary computer; and because Linux is free, it doesn't cost anything to equip these PCs with an operating system. While proprietary parallel computer companies are still in business, their computers primarily occupy a niche at the ultra-high-performance end of the spectrum. The majority of parallel computing nowadays takes place on commodity hardware.

Since 1993, the TOP500 List (at http://www.top500.org) has been tracking the 500 fastest supercomputers in the world, as measured by a standard benchmark (the LINPACK linear algebra benchmark). In the June 1993 TOP500 List, only 104 of the top 500 supercomputers (21%) used commodity CPU chips—Intel i860s and Sun SuperSPARCs. The rest of the top 500 (79%) used proprietary CPUs. In contrast, the June 2008 TOP500 list shows 498 of the top 500 supercomputers using commodity CPU chips from (in alphabetical order) AMD, IBM, and Intel. By the way, in June 1993, 97 of the top 500 supercomputers were still single-CPU systems. In June 2008, *all* the top 500 supercomputers were parallel computers of one kind or another, with anywhere from 80 to over 200,000 processors.

In the transition to commodity hardware, parallel programming also shifted away from using proprietary programming languages to using standard, non-parallel programming languages, chiefly Fortran, C, and C++, coupled with standard parallel programming libraries. The Parallel Virtual Machine (PVM) library for programming cluster computers was first released in 1989. The Message Passing Interface (MPI) standard, also for programming cluster computers, was first released in 1994. The OpenMP standard for multithreaded parallel programming was first released in 1997. Like Linux, free versions of PVM, MPI, and OpenMP are widely available. Because these are hardware-independent standards, parallel programs written on one machine can be easily ported to another machine. The majority of parallel programming nowadays is done using a standard language and library.

Because parallel computing with commodity hardware and software is now firmly established, this book will focus on building parallel programs for commodity parallel computers. To build good parallel programs requires understanding the characteristics of parallel computer hardware and software that influence parallel program design. Next, we'll look at these characteristics and at the prevalent parallel computer architectures.

2.2 CPU Hardware

To help achieve the goal of ever-increasing computer performance (and sales), **central processing units (CPUs)** have become bewilderingly complex. A modern CPU may employ architectural features such as these:

- **Pipelined architecture**. While one instruction is being fetched from memory, another already-fetched instruction is being decoded, and several more already-decoded instructions are in various stages of execution. With multiple instructions in process at the same time in different stages of the pipeline, the CPU's effective computation speed increases.
- **Superscalar architecture**. The CPU has several functional units, each capable of executing a different class of instructions—an integer addition unit, an integer multiplication unit, a floating-point unit, and so on. If several instructions use different functional units and do not have other dependencies among each other, the CPU can execute all the instructions at once, increasing the CPU's effective speed.
- **Instruction reordering**. To utilize the CPU's pipeline and functional units to the fullest, the CPU may issue instructions in a different order from the order they are stored in memory, provided the results turn out the same.

High-level programming languages hide most of this architectural complexity from the programmer. The hardware itself, perhaps aided by the high-level language compiler or (in the case of Java) the virtual machine, takes care of utilizing the CPU's architectural features. A Java or C program, for example, does not have to be rewritten if it is going to be run on a superscalar CPU rather than a CPU with just one functional unit. However, there are two CPU architectural features that *do* make a difference in the way high-level language programs are written: cache memories, and symmetric multiprocessors.

Cache memories. As CPU speeds outpaced memory speeds, computer architects added a **cache memory** to avoid making a fast CPU wait for a slow main memory (Figure 2.2).

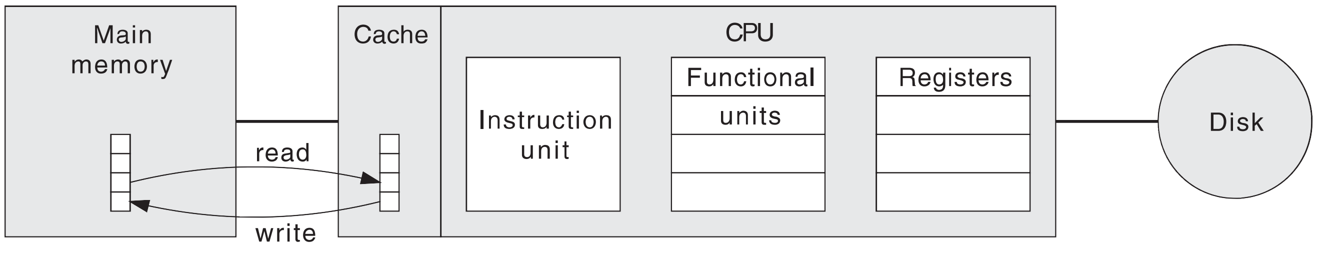

Figure 2.2 Computer with cache memory

The main memory is large, typically gigabytes or hundreds of megabytes, but slow. The cache memory is much smaller than main memory, typically a few megabytes, but is also much faster than main memory. That is, it takes much less time for the CPU to read or write a word in cache memory than in main memory.

With the cache in place, when the CPU reads a word at a certain address, the CPU first checks whether the desired word has been loaded into the cache. If it has not—a **cache miss**—the CPU loads the entire **cache line** containing the desired word from main memory into the cache; then the CPU reads the desired word from the cache. The cache line size depends on the processor; 64 to 128 bytes is typical. Thereafter, when the CPU reads the same word, or reads another word located in the same cache line, there is a **cache hit**; the CPU reads the word directly from the cache and does not need to load it from main memory. Thus, if the **cache hit ratio**—the fraction of all memory accesses that are cache hits—is large, the CPU will read data words at nearly the speed of the fast cache memory and will seldom have to wait for the slow main memory.

As the CPU continues to load cache lines from main memory into the cache, eventually the cache becomes full. At the next cache miss, the CPU must replace one of the previously loaded cache lines with the new cache line. The cache's **replacement policy** dictates which cache line to replace. Various replacement policies are possible, such as replacing a *randomly chosen* cache line, or replacing the cache line that has not been accessed for the longest amount of time (the *least recently used* cache line).

When the CPU writes a word at a certain address, the CPU must load the relevant cache line from main memory if there is a cache miss, then the CPU can replace the contents of the desired word in the cache with the new value. Subsequent reads of that word will retrieve the new value from the cache. However, after a write, the contents of the word in the cache do not match the contents of the word in main memory; the cache line is said to be **dirty**. The cache's **write policy** dictates what to do about a dirty cache line. A *write-through cache* copies the dirty cache line to main memory immediately. A *write-back cache* copies the dirty cache line to main memory only when the cache line is being replaced.

The cache has a profound effect on program performance. A program's **working set** consists of all the memory locations the program is currently accessing, both locations that contain instructions and locations that contain data. If the program's working set fits entirely within the cache, the CPU will be able to execute the program as fast as it possibly can. This is often possible when the bulk of the program's time is spent in a tight loop operating on a data structure smaller than the cache. To the extent that the program's working set does not fit in the cache, the program's performance will be reduced. In this case, the name of the game is to design the program to minimize the number of inevitable cache misses.

Some CPUs are so fast that even the cache is a performance bottleneck. Such CPUs use a **multilevel cache** (Figure 2.3).

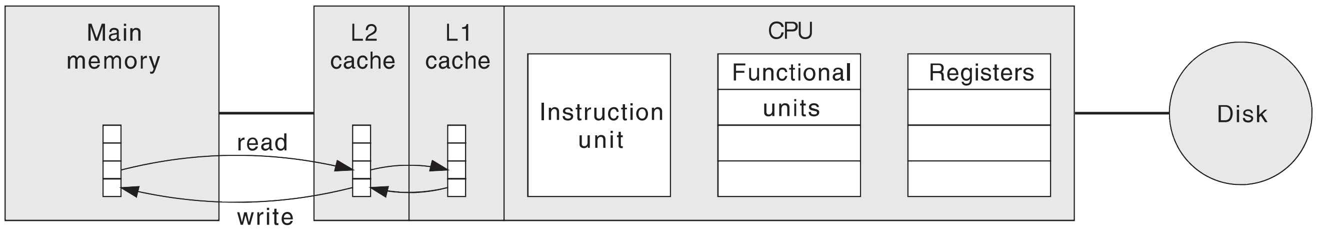

Figure 2.3 Computer with multilevel cache

A **level-1 (L1) cache** sits between the CPU and the **level-2 (L2) cache** (formerly the only cache). The L1 cache is even faster than, and smaller than, the L2 cache. The L1 cache bears the same relationship to the L2 cache as the L2 cache bears to main memory. From the programmer's point of view, whether the cache has multiple levels is less important than the cache's existence. The name of the game is still to design the program to minimize the number of cache misses.

Symmetric multiprocessors. To achieve performance gains beyond what is possible on a single CPU, computer architects replicated the CPU, resulting in a **symmetric multiprocessor** (Figure 2.4). It is called "symmetric" because all the processors are identical. Each processor is a complete CPU with its own instruction unit, functional units, registers, and cache. All the processors share the same main memory and peripherals. The computer achieves increased performance by running multiple threads of execution simultaneously, one on each processor.

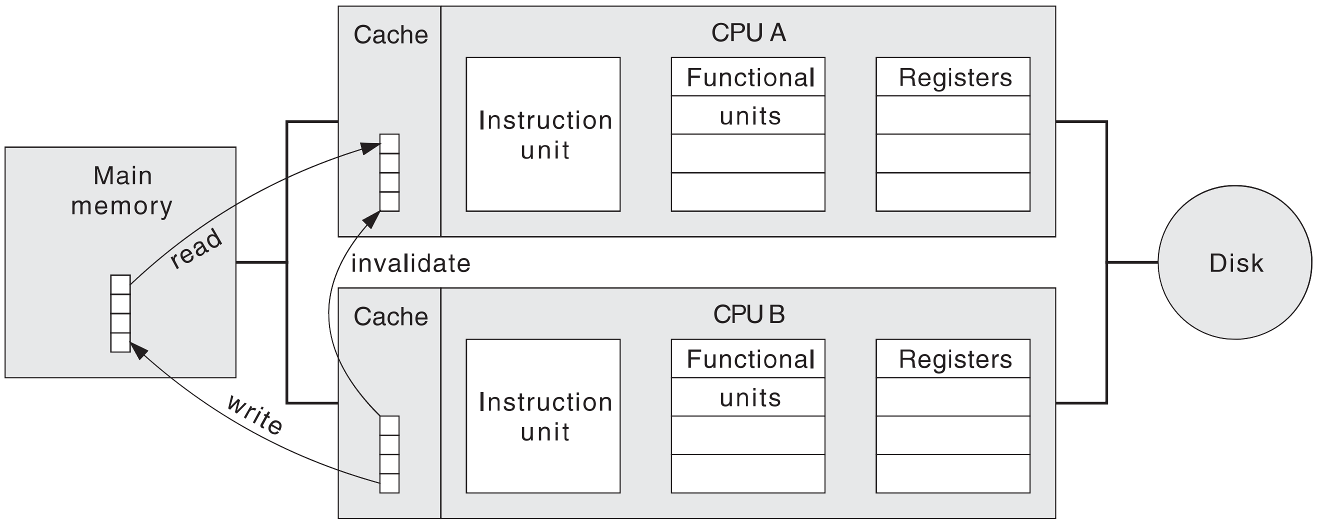

Figure 2.4 Symmetric multiprocessor

Going to multiple processors, however, affects the caches' operation. Suppose CPU A reads the word at a certain address *x,* so that a copy of the relevant cache line is loaded from main memory into CPU A's cache. Suppose CPU B now writes a new value into the word at address *x*. CPU B's cache line is written back to main memory. However, CPU A's cache line no longer has the correct contents. The CPUs use a **cache coherence protocol** to bring the caches back to a consistent state. One popular cache coherence protocol uses **invalidation**. When CPU B writes its value into address *x*, CPU B sends an "invalidate" signal to tell CPU A that the contents of address *x* changed. In response, CPU A changes its cache state to say that the cache line containing address *x* does not reside in the cache. This is called "invalidating" the cache line. The next time CPU A reads address *x*, CPU A will see that the cache line containing address *x* is not loaded, CPU A will reload the cache line from main memory, and CPU A will retrieve the value written by CPU B.

Note that writing a word in one CPU can slow down another CPU, because the other CPU has to re-read the word's cache line from slow main memory. This **cache interference** effect can reduce a parallel program's performance, and we will return to the topic of cache interference in Chapter 9.

Symmetric multiprocessor computers originally used a separate CPU chip for each processor (thread). However, as the number of transistors that could fit on a chip continued to increase, computer architects started to contemplate running more than one thread on a single chip. For example, instead of simultaneously issuing multiple instructions from a single instruction stream—a single thread—to multiple functional units, why not issue multiple instructions from *multiple* threads simultaneously to the functional units? Doing this for, say, two threads requires two instruction units, one to keep track of each thread's instruction stream. The result is called a **hyperthreaded CPU** (Figure 2.5).

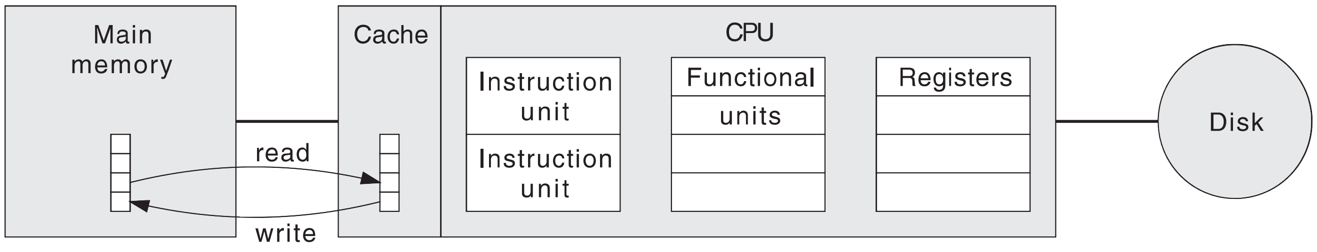

Figure 2.5 Computer with hyperthreaded CPU

Two instruction units are by no means the limit. Some of today's fastest supercomputers use **massively multithreaded processors (MMPs)** that can handle hundreds of simultaneous threads. As soon as one thread stalls, perhaps because it has to wait for data to be loaded from main memory into the cache, the CPU can instantly switch to another thread and keep computing.

As transistor densities continued to increase, it became possible to replicate the whole processor, not just the instruction unit, on a single chip. In other words, it became possible to put a symmetric multiprocessor on a chip. Such a chip is called a **multicore CPU**. Alternatively, the name may refer to the number of processors on the chip: a **dual-core CPU**; a **quad-core CPU**; and so on. Multicore chips can themselves be aggregated into symmetric multiprocessor systems, such as a four-processor computer comprising two dual-core CPU chips.

It used to be that you could increase an application program's performance simply by trading in your old PC for a new one with a faster CPU chip. That won't necessarily work any longer. Now that chips have become hyperthreaded or multicore, your new PC may very well have a multicore CPU with the same clock speed as, or even a slower clock speed than, your old PC. If your application is single-threaded, as many are, it can run only on one CPU of the multicore chip and thus may run *slower* on your new PC! Until applications are redesigned as multithreaded programs—*parallel* programs—users will not see much of a performance improvement when running applications on the latest multicore PCs.

Having examined the salient features of individual CPUs, we next look at different ways to combine multiple CPUs to make a complete parallel computer.

2.3 SMP Parallel Computers

There are three principal parallel computer architectures using commodity hardware: SMPs, clusters, and hybrids. There is also a variant called a computing grid. Parallel coprocessors using commodity graphics chips have also been introduced.

A **shared memory multiprocessor (SMP)** parallel computer is nothing more than a symmetric multiprocessor system (Figure 2.6). Each processor has its own CPU and cache. All the processors share the same main memory and peripherals.

A parallel program running on an SMP parallel computer (Figure 2.7) consists of one process with multiple threads, one thread executing on each processor. The process's program and data reside in the shared main memory. Because all threads are in the same process, all threads are part of the same **address space**, so all threads access the same program and data. Each thread performs its portion of the computation and stores its results in the shared data structures. If the threads need to communicate or coordinate with each other, they do so by reading and writing values in the shared data structures.

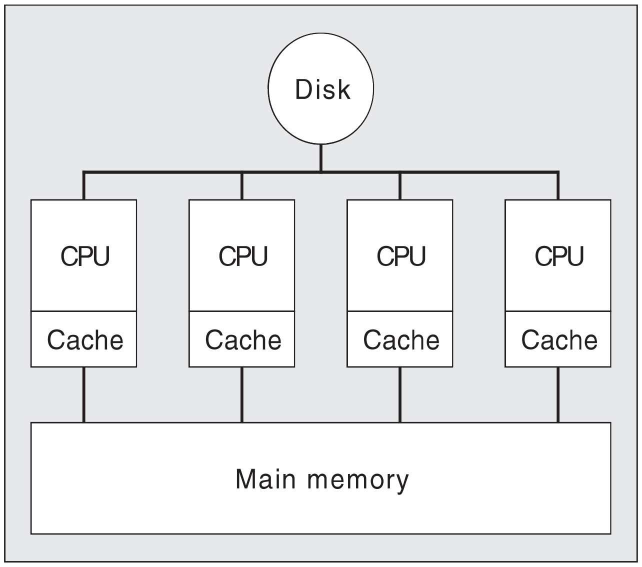

Figure 2.6 SMP parallel computer

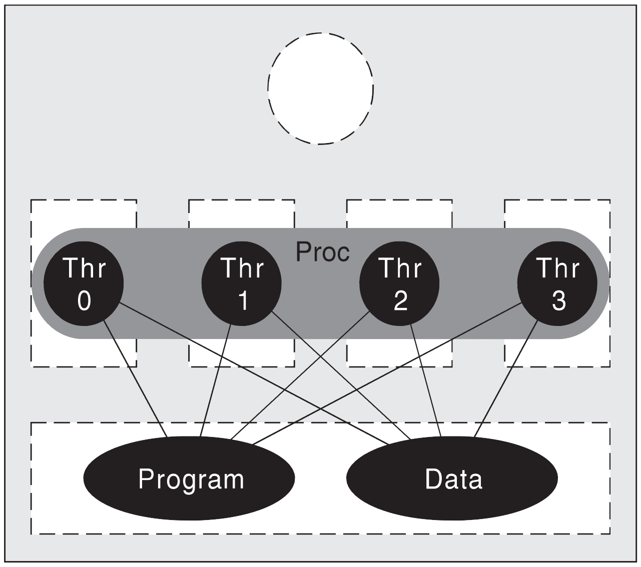

Figure 2.7 SMP parallel program

2.4 Cluster Parallel Computers

A **cluster** parallel computer consists of multiple interconnected processor nodes (Figure 2.8). There are several **backend processors** that carry out parallel computations. There is also typically a separate **frontend processor**; users log into the frontend to compile and run their programs. There may be a shared file server. Each backend has its own CPU, cache, main memory, and peripherals, such as a local disk drive. Each processor is also connected to the others through a dedicated high-speed **backend network**. The backend network is used only for traffic between the nodes of the cluster; other network traffic, such as remote logins over the Internet, goes to the frontend. Unlike an SMP parallel computer, there is no global shared memory; each backend can access only its local memory. The cluster computer is said to have a **distributed memory**.

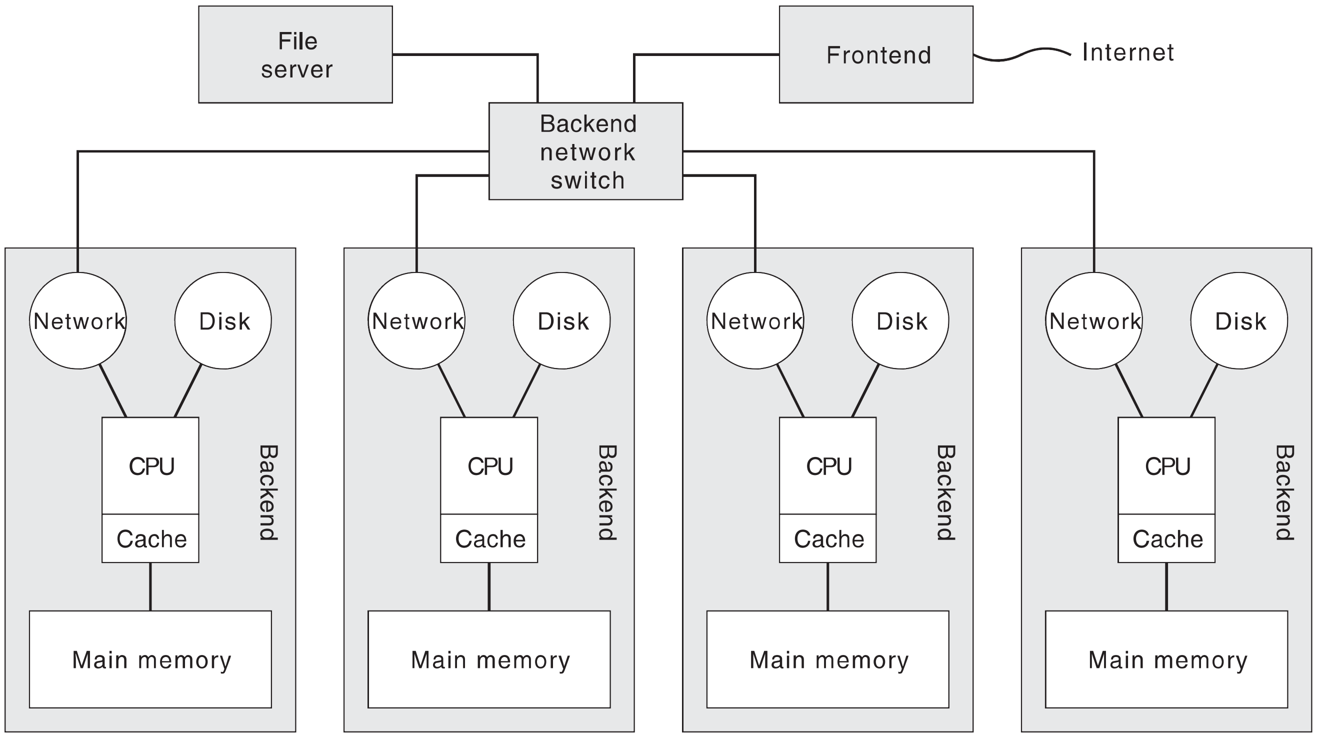

Figure 2.8 Cluster parallel computer

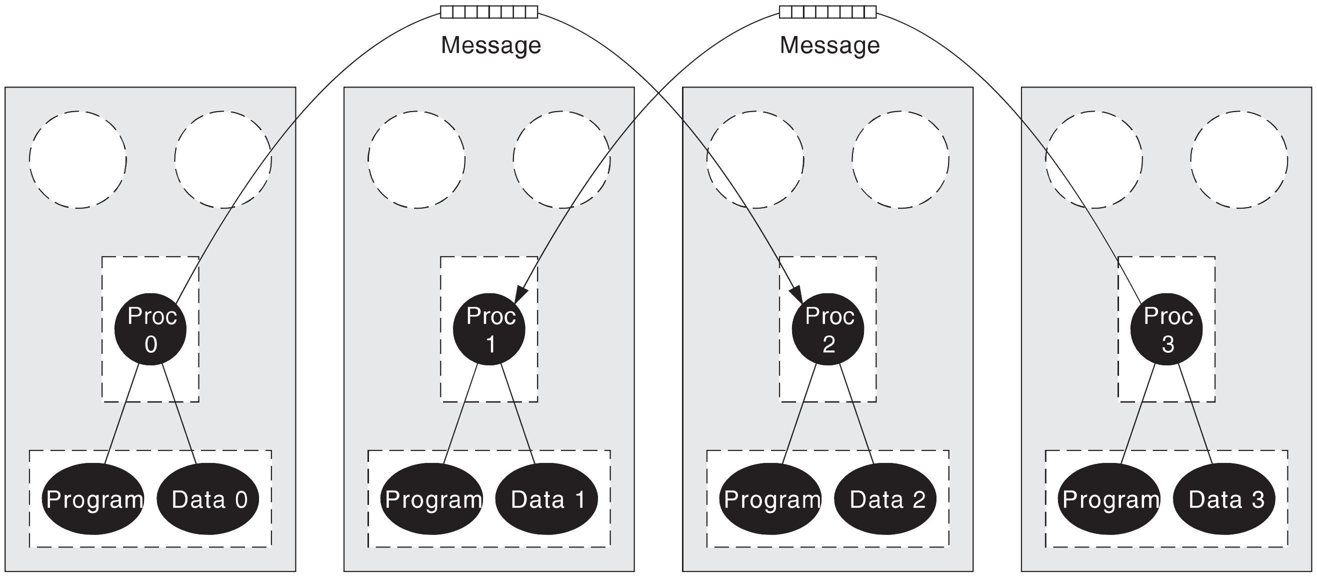

Figure 2.9 Cluster parallel program

A parallel program running on a cluster parallel computer (Figure 2.9) consists of multiple processes, one process executing on each backend processor. Each process has its own, separate address space. All processes execute the same program, a copy of which resides in each process's address space in each backend's main memory. The program's data is divided into pieces; a different piece resides in each process's address space in each backend's main memory. Each process performs its portion of the computation and stores its results in the data structures in its own local memory. If one process needs a piece of data that resides in another process's address space, the process that owns the data sends a **message** containing the data through the backend network to the process that needs the data. The processes may also exchange messages to coordinate with one another, without transferring data. Unlike an SMP parallel program where the threads can simply access shared data in the one process's address space, in a cluster parallel program, the processes must be explicitly coded to send and receive messages.

To run a parallel program on a cluster, you typically log into the cluster's frontend processor and launch the program in a process on the frontend, like any other program. Under the hood, the frontend process uses a cluster middleware library to start a backend process on each backend processor, load a copy of the program into each backend's memory, initialize connections for message passing through the backend network, and commence execution of each backend process. At program completion, when all the backend processes have terminated, the frontend process also terminates.

For maximum performance, it's not enough to equip a cluster with fast backend processors. It's also important for the cluster's backend network to have three characteristics: low latency, high bandwidth, and high bisection bandwidth.

Latency refers to the amount of time needed to start up a message, regardless of the message's size; it depends on the hardware and software protocols used on the network. **Bandwidth**, measured in bits per second, is the rate at which data is transmitted once a message has started. The time to transfer a message is the latency, plus the message size divided by the bandwidth. A small latency and a large bandwidth will minimize each message's transfer time, thus reducing the cluster parallel program's running time.

Bisection bandwidth, also measured in bits per second, refers to the total rate at which data can be transferred if half the nodes in the cluster are sending messages to the other half. In other words, the network is split down the middle—bisected—and each node on one side of the split sends data as fast as possible to a different node on the other side of the split. As we will see in Part III, some cluster parallel programs do in fact send messages from half the processes to the other half at the same time. Ideally, in an N-node cluster, the network's bisection bandwidth would be $N/2$ times the bandwidth on a single link. Depending on how the backend network is built, however, the bisection bandwidth may be less than the ideal, which in turn may reduce the cluster parallel program's performance.

Several commodity off-the-shelf technologies are used for backend networks in cluster parallel computers. The available alternatives fall into two categories: Ethernet, and everything else.

Ethernet, due to its ubiquity, is the least-expensive alternative. Ethernet interface cards, switches, and cables that support a bandwidth of 1 gigabit per second (1 Gbps), or 1×10^9 bits per second, are readily available. Although more expensive, 10 Gbps Ethernet equipment is also available, and 100 Gbps Ethernet is on the horizon. An Ethernet *switch* usually has a large bisection bandwidth. (An Ethernet *hub* does not; you should never use a hub to build a cluster backend network.) Platform-independent programs that communicate over Ethernet are easily written using the standard socket application program interface (API) and the Internet standard TCP/IP protocols. However, Ethernet has a much higher latency—around 150 microseconds (150 μsec), or 150×10^{-6} seconds—than the alternatives, especially when using TCP/IP.

Other interconnection technologies used in cluster parallel computers, such as **InfiniBand**, **Scalable Coherent Interface (SCI)**, and **Myrinet**, are all more or less the same in their gross characteristics. They all support higher bandwidths than Ethernet (up to about 100 Gbps), much lower latencies than Ethernet (in the single microsecond range), and high bisection bandwidths. However, not being nearly as widespread as Ethernet, they are all more expensive. They also tend to require the use of technology-specific software libraries to achieve their full performance. While TCP/IP can be layered on top of these technologies, thus allowing platform-independent programs to run on these technologies, layering TCP/IP increases the latency due to the TCP/IP protocol overhead.

Ultra-high-performance parallel supercomputers often use interconnect technologies such as InfiniBand, SCI, or Myrinet because of their higher bandwidth and lower latency. Mundane parallel computers usually use Ethernet because of its lower cost and platform-independent programming support.

2.5 Hybrid Parallel Computers

A **hybrid** parallel computer is a cluster in which each backend processor is an SMP machine (Figure 2.10). In other words, it is a combination, or hybrid, of cluster and SMP parallel computers. A hybrid parallel computer has both shared memory (within each backend) and distributed memory (between backends). Eventually, all commodity clusters will be hybrid parallel computers because multicore PCs are becoming popular and single-core PCs are becoming harder to find.

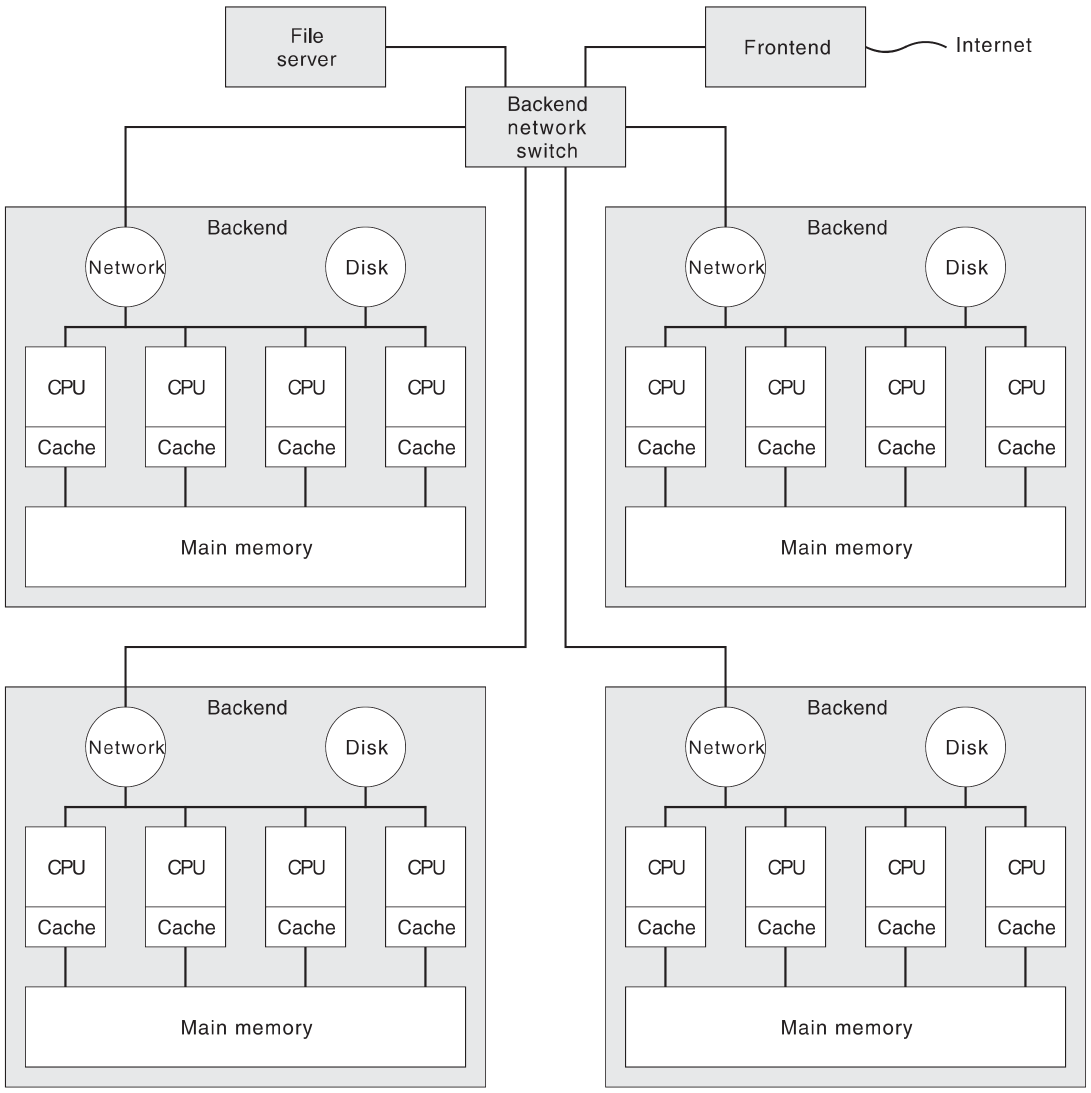

Figure 2.10 Hybrid parallel computer

A hybrid parallel computer is programmed using a combination of cluster and SMP parallel programming techniques (Figure 2.11). Like a cluster parallel computer, each backend runs a separate process with its own address space. Each process executes a copy of the same program, and each process has a portion of the data. Like an SMP parallel computer, each process in turn has multiple threads, one thread running on each CPU. Threads in the same process share the same address space and can access their own shared data structures directly. Threads in different processes must send messages to each other to transfer data.

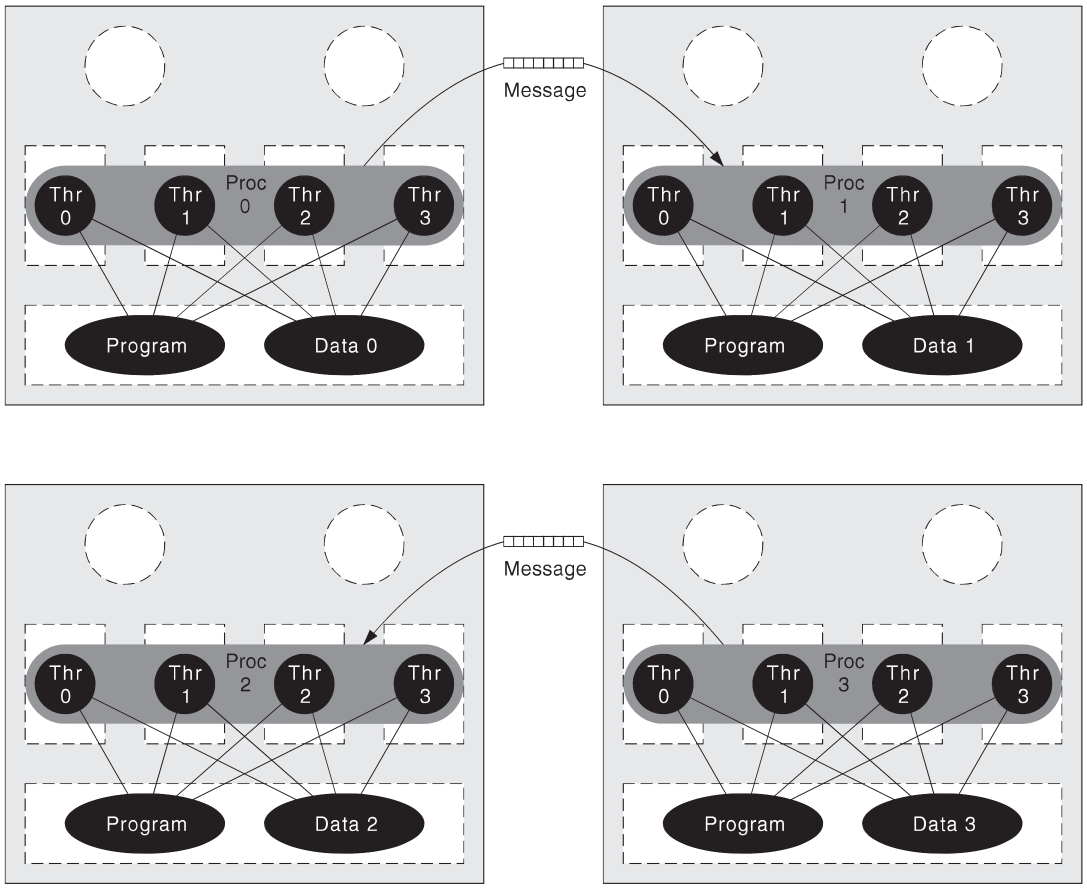

Figure 2.11 Hybrid parallel program

2.6 Computing Grids

Architecturally, a **computing grid** is the same as a cluster parallel computer, except that the processors are not all located in the same place and are not all connected to a common, dedicated backend network. Instead, the processors are located at diverse sites and are connected through a combination of local area networks and the Internet (Figure 2.12).

Often, a grid is set up by a consortium of companies, universities, research institutions, and government agencies—the member organizations contributing computers to the grid. The Open Science Grid (OSG), for example, is a grid devoted to large-scale scientific computation, with thousands of processors located at 85 institutions in Brazil, Canada, England, Germany, Korea, Taiwan, and the United States.

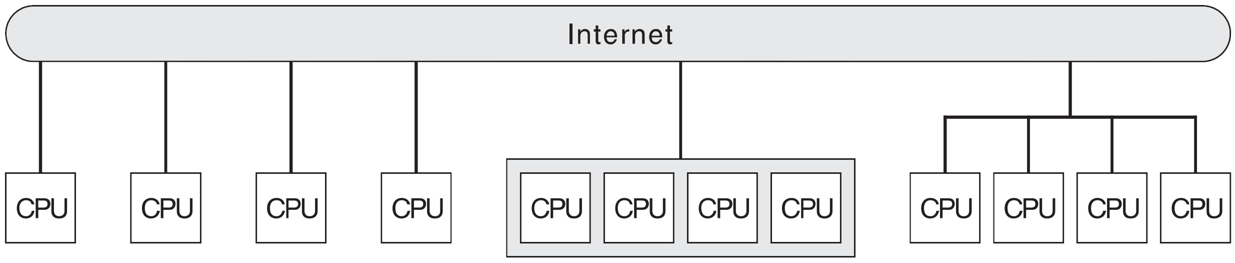

Figure 2.12 Computing grid with single processors, an SMP, and a cluster

Other grids are based on "volunteer computing." Users download a special client program to their desktop PCs. The client program runs as a low-priority background process, or sometimes as a screen saver. When the PC becomes idle, the client program starts up and executes a portion of a parallel computation, communicating with other nodes over the Internet. Projects using volunteer computing grids include SETI@home, the Great Internet Mersenne Prime Search (GIMPS), and dozens of others. The Berkeley Open Infrastructure for Network Computing (BOINC) is a general framework for developing parallel programs that run on volunteer computing grids.

Computing grids are programmed in the same way as cluster parallel computers, with multiple processes running on the various grid machines. However, a parallel program that performs well on a cluster is not necessarily well suited for a grid. The Internet's latency is orders of magnitude larger, and the Internet's bandwidth is orders of magnitude smaller, than a typical cluster backend network. Because some of the computers running the grid program may be connected through the Internet, the average message takes a lot longer to send on a grid than on a cluster. Thus, parallel programs that require intensive message passing do not perform well on a grid. Problems best suited for a grid are those that can be divided into many pieces that are computed independently with little or no communication.

In this book, we will study how to build parallel programs for *tightly coupled* processors: SMP parallel computers where all processors use a single shared memory; and cluster and hybrid parallel computers where all processors are connected to the same high-speed backend network. Parallel programming for *loosely coupled* computing grids is beyond the scope of this book.

2.7 GPU Coprocessors

CPU, memory, and Ethernet chips are not the only chips that have become commodities. Driven by the market's insatiable appetite for ever-higher-performing graphics displays on PCs and game consoles, **graphics processing unit (GPU)** chips have also become commonplace.

Acting as a **coprocessor** to the main CPU, the GPU is a specialized chip that handles graphics rendering calculations at very high speeds. The CPU sends high-level commands to the GPU to draw lines and fill shapes with realistic shading and lighting; the GPU then calculates the color of each pixel and drives the display. Because the pixels can be computed independently, the GPU typically has multiple processing cores and calculates multiple pixels in parallel.

Programmers have realized that GPUs can be used to do parallel computations other than pixel rendering, and have even coined an acronym for it: **GPGPU**, or General Purpose computation on Graphics Processing Units. In response, GPU vendors have repackaged their graphics cards as general-purpose massively parallel coprocessors, complete with on-board shared memory and with APIs for parallel programming on the GPU. A GPU coprocessor card transforms a regular PC into what marketers call a “desktop supercomputer” (Figure 2.13). The low cost of the commodity GPU chips makes a GPU coprocessor an attractive alternative to general-purpose multicore CPUs and clusters.

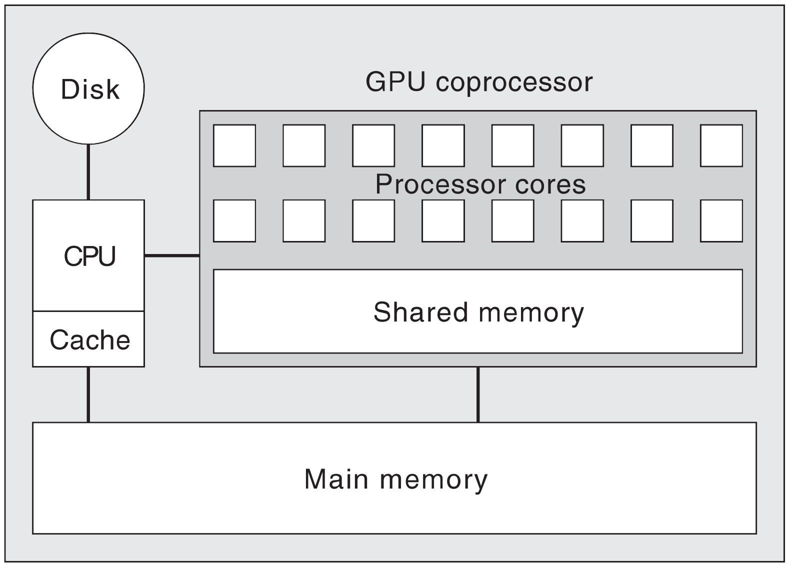

Figure 2.13 Parallel computer with GPU coprocessor

A GPU’s instruction set is usually more limited than a regular CPU’s. For example, a GPU may support single-precision floating-point arithmetic, but not double precision. Also, the GPU’s cores typically must all perform identical operations simultaneously, each core operating on its own data items. For these reasons, current GPU coprocessors cannot run arbitrary parallel programs. However, GPU coprocessors excel at running “inner loops,” where the same statements are performed on every element of an array or matrix at very high speed in parallel. Thus, a GPU parallel program usually consists of regular code executed on the main CPU with a computation-intensive section executed on the GPU. Unfortunately, space limitations do not permit covering GPU parallel programming in this book.

2.8 SMPs, Clusters, Hybrids: Pros and Cons

Why are there three prevalent parallel computer architectures? Why not use the same architecture for all parallel computers? The reason is that each architecture is best suited for certain kinds of problems and is not well suited for other kinds of problems.

An SMP parallel computer is well suited for a problem where there are data dependencies between the processors—where each processor produces results that are used by some or all of the other processors. By putting the data in a common address space (shared memory), each thread can access every other thread’s results directly at the full speed of the CPU and memory. A cluster parallel computer running such a program would have to send many messages between the processors. Because sending a message is orders of magnitude slower than accessing a shared memory location, even with high-speed interconnects such as InfiniBand, SCI, and Myrinet, an SMP parallel computer would outperform a cluster parallel computer on this problem.

Conversely, a cluster parallel computer is well suited for a problem where there are few or no data dependencies between the processors—where each processor can compute its own results with little or no communication with the other processors. The more communication needed, the poorer a cluster parallel program will perform compared to an SMP parallel program.

On the other hand, two limitations are encountered when trying to scale up to larger problem sizes on an SMP parallel computer. First, there is a limit on the physical memory size. A 32-bit CPU can access at most 4 gigabytes (2^{32} bytes) of physical memory. While a 64-bit CPU can theoretically access up to 16 million terabytes (2^{64} bytes) of physical memory, it will be a long time before any single CPU has a physical memory that large. Although virtual memory lets a process access a larger address space than physical memory, the performance of a program whose instructions and data do not fit in physical memory would be severely reduced due to swapping pages back and forth between physical memory and disk. Second, there is a limit on the number of CPUs that can share the same main memory. The more CPUs there are, the more circuitry is needed to coordinate the memory transactions from all the CPUs and to keep all the CPUs' caches coherent. Thus, an SMP parallel program's problem size cannot scale up past the point where its data no longer fits in main memory, and its speedup or sizeup cannot scale up past a certain number of CPUs.

With a cluster parallel computer, there are no limits on scalability due to main memory size or number of processors. On a *K*-node cluster, the maximum total amount of memory is *K* times the maximum on one node. Thus, if you need more memory, more speedup, or more sizeup, just add more nodes to the cluster. However, because message latency tends to increase as the number of nodes on the network increases, causing program performance to decrease, a cluster cannot keep growing forever. Still, commodity clusters with hundreds and even thousands of nodes have been built; the largest commodity SMPs have dozens of nodes at most.

Like a cluster, a hybrid parallel computer can be scaled up to larger problem sizes by adding more nodes. In addition, a hybrid parallel computer is especially well suited for a problem that can be broken into chunks having few or no data dependencies between chunks, but that can have significant data dependencies within each chunk. The chunks can be computed in parallel by separate processes running on the cluster nodes and passing messages among each other. Within each chunk, the results can be computed in parallel by separate threads running on the node's CPUs and accessing data in shared memory. Many of the supercomputers in the TOP500 List are commodity hybrid parallel computers.

Any parallel program can be implemented to run on an SMP, cluster, or hybrid parallel computer. (For some of the example programs in this book, we will look at all three variations.) Which kind of parallel computer, then, should you use to solve your high-performance computing problem—assuming your local computer center even offers you a choice? The answer is to use the kind of parallel computer that gives the best performance on your problem. However, because so many factors influence performance, the only way to know for sure is to implement the appropriate versions of the program and try them on the available parallel computers. As we study how to design and code parallel programs, we will also discuss how the program's design influences the program's performance, and we will see how certain kinds of parallel programs perform better on certain kinds of parallel computers.

2.9 Parallel Programming Libraries

It is perfectly possible to write parallel programs using a standard programming language and generic operating system kernel functions. You can write a multithreaded program in C using the standard POSIX thread library (Pthreads), or in Java using Java's built-in Thread class. If you run this program on an SMP parallel computer, each thread will run on a different processor simultaneously, yielding a parallel speedup. You can write a multiprocess program in C using the standard socket API to communicate between processes, or in Java using Java's built-in java.net.Socket class. If you run copies of this process on the backend processors of a cluster parallel computer, the simultaneously running processes will yield a parallel speedup.

However, parallel programs are generally not written this way, for two reasons. First, some programming languages popular in domains that benefit from parallel programming—such as Fortran for scientific computing—do not support multithreaded programming and network programming as well as other languages. Second, using low-level thread and socket libraries increases the effort needed to write a parallel program. It takes a great deal of effort to write the code that sets up and coordinates the multiple threads of an SMP parallel program, or to write the code that sets up network connections and sends and receives messages for a cluster parallel program. Parallel program users are interested in solving problems in their domains, such as searching a massive DNA sequence database or calculating the motion of stars in a galaxy, not in writing thread or network code. Indeed, many parallel program users may lack the expertise to write thread or network code.

Instead, to write a parallel program, you use a **parallel programming library**. The library encapsulates the low-level thread or network code that is the same in any parallel program, presenting you with easy-to-use, high-level parallel programming abstractions. You can then devote most of your parallel programming effort to solving your domain problem using those abstractions.

OpenMP is the standard library for SMP parallel programming. OpenMP supports the Fortran, C, and C++ languages. By inserting special OpenMP **pragmas** into the source code, you designate which sections of code are to be executed in parallel by multiple threads. You then run the annotated source code through a special OpenMP compiler. The OpenMP compiler looks at the OpenMP pragmas, *rewrites* your source code to add the necessary low-level threading code, and compiles your now-multithreaded program as a regular Fortran, C, or C++ program. You then run your program as usual on an SMP parallel computer. See Appendix A for further information about OpenMP.

The **Message Passing Interface (MPI)** is the standard library for cluster parallel programming. Like OpenMP, MPI supports the Fortran, C, and C++ languages. Unlike OpenMP, MPI requires no special compiler; it is just a subroutine library. You write your parallel program like any regular program, calling the MPI library routines as necessary to send and receive messages, and compile your program as usual. To run your program on a cluster parallel computer, you execute a special MPI **launcher** program. The launcher takes care of starting a process to run your compiled executable program on each backend processor. The MPI library routines your program calls then take care of all the details of setting up network connections between processes and passing messages back and forth. See Appendix B for further information about MPI.

As already mentioned, hybrid parallel computers are becoming popular, due to the wide availability of multicore PCs. However, as yet, there are no standard libraries for hybrid parallel programming. OpenMP has no routines for message passing. MPI has no capabilities for executing sections of code in parallel in multiple threads. Writing a hybrid parallel program using both OpenMP and MPI is not guaranteed to work, because the MPI standard does not require all MPI implementations to support multithreaded programs (although an MPI implementation is allowed to do so). While hybrid parallel programs can be written solely using MPI by running a separate *process* (rather than a thread) on each CPU of each node, sending messages between different processes' address spaces on the same node often yields poorer performance than simply sharing the same address space among several threads.

In addition, neither the official OpenMP standard nor the official MPI standard supports the Java language. Yet Java is becoming the language that most computing students learn. While several unofficial versions of MPI and OpenMP in Java have appeared, none can be considered a standard, and none are designed for hybrid parallel programming.

In this book, we will use the **Parallel Java Library** to learn how to build parallel programs. Parallel Java includes both the multithreaded parallel programming capabilities of OpenMP and the message-passing capabilities of MPI, integrated in a single library. Thus, Parallel Java is well suited for hybrid parallel programming as well as SMP and cluster parallel programming. Parallel Java also includes its own middleware for running parallel programs on a cluster. Because the library is written in 100% Java, Parallel Java programs are portable to any machine that supports Java (JDK 1.5). Appendices A and B compare and contrast Parallel Java with OpenMP and MPI.

2.10 For Further Information

On the history of parallel computers:

- G. Wilson. A chronology of major events in parallel computing. University of Toronto Computer Systems Research Institute Technical Report CSRI–312, December 1994. ftp://ftp.cs.toronto.edu/csrg-technical-reports/312/csri312.ps

On Beowulf:

- T. Sterling, D. Becker, D. Savarese, J. Dorband, U. Ranawake, and C. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing (ICPP 1995)*, 1995, volume 1, pages 11–14.
- D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: harnessing the power of parallelism in a Pile-of-PCs. In *Proceedings of the 1997 IEEE Aerospace Conference*, 1997, volume 2, pages 79–91.
- Beowulf.org Web site. http://www.beowulf.org/

On the Stone SouperComputer (and the tale of "Stone Soup"):

- W. Hargrove, F. Hoffman, and T. Sterling. The do-it-yourself supercomputer. *Scientific American*, 265(2):72–79, August 2001.
- The Stone SouperComputer. http://www.extremelinux.info/stonesoup/

On the Parallel Virtual Machine (PVM) library:

- V. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency Practice and Experience*, 2(4):315–339, December 1990.
- A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/pvm_home.html

On the official MPI standard:

- Message Passing Interface Forum Web Site. http://www.mpi-forum.org/
- Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. June 12, 1995. (MPI Version 1.1) http://www.mpi-forum.org/docs/mpi-11.ps
- Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. July 18, 1997. (MPI Version 2.0) http://www.mpi-forum.org/docs/mpi-20.ps

A comparison of PVM and MPI:

- G. Geist, J. Kohl, and P. Papadopoulos. PVM and MPI: a comparison of features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

On the official OpenMP standard:

- OpenMP.org Web Site. http://openmp.org/wp/
- OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*. May 2008. http://www.openmp.org/mp-documents/spec30.pdf

On the TOP500 supercomputer list and the LINPACK benchmark:

- TOP500 Supercomputer Sites. http://www.top500.org/
- LINPACK. http://www.netlib.org/linpack/
- LINPACK Benchmark—Java Version. http://www.netlib.org/benchmark/linpackjava/

On the Open Science Grid:

- Open Science Grid. http://www.opensciencegrid.org/

On volunteer computing grids:

- Distributed Computing Projects directory.
  http://www.distributedcomputing.info/
- SETI@home. http://setiathome.berkeley.edu/
- GIMPS. http://www.mersenne.org/
- Berkeley Open Infrastructure for Network Computing.
  http://boinc.berkeley.edu/

On parallel computing with GPUs:

- GPGPU Web Site. http://www.gpgpu.org/
- N. Goodnight, R. Wang, and G. Humphreys. Computation on programmable graphics hardware. *IEEE Computer Graphics and Applications*, 25(5):12–15, September/October 2005.
- J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
- R. Fernando, editor. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Education, 2004.
- M. Pharr, editor. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Pearson Education, 2005.
- H. Nguyen, editor. *GPU Gems 3*. Addison-Wesley, 2007.

On the Parallel Java Library:

- A. Kaminsky. Parallel Java: a unified API for shared memory and cluster parallel programming in 100% Java. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007),* March 2007.
- Parallel Java Library (including downloads).
  http://www.cs.rit.edu/~ark/pj.shtml
- Parallel Java documentation.
  http://www.cs.rit.edu/~ark/pj/doc/index.html