Chapter 1
The Parallel Landscape

Part I. Preliminaries

Chapter 1. The Parallel Landscape

Part II. Tightly Coupled Multicore

Part III. Loosely Coupled Cluster

Part IV. GPU Acceleration

Part V. Map-Reduce
Parallel computing is concerned with designing computer programs having two characteristics: they run on *multiple processors*, or *cores*, and all the cores cooperate with each other to solve a *single problem*.

Both characteristics are necessary for a program to be considered a parallel program. A web server, like the one that runs Google’s web site, typically runs on a multicore server machine, or even a group of several multicore server machines, so as to process hundreds or thousands of web page requests each second simultaneously with high throughput. The web server thus displays the first characteristic of parallel computing. However, the web server’s cores are working on *different* problems, namely the different user’s web page requests. The web server’s cores are not cooperating with each other to solve the *same* problem. So the web server does not display the second characteristic of parallel computing. I would call a web server an example of *distributed* computing, not parallel computing. It’s a subtle distinction, but a crucial one. In this book I am going to be discussing parallel computing, not distributed computing.

Google does, however, do other computations that *are* examples of parallel computing—namely, the big data analytics that convert the raw web pages swept up by Google’s web crawlers into the query indexes that let Google speedily retrieve web pages that best match users’ searches. Here, Google uses *multiple cores* cooperating to solve the *single problem* of converting enormous quantities of web page text into enormous query indexes. In fact, Google was the leader in popularizing the *map-reduce* paradigm of parallel computing on massive data sets. Google, and many other major Internet sites, also use map-reduce parallel computing to convert enormous quantities of information gathered from us users—the queries we do, the web pages we visit, the contents of our Gmail emails, Facebook posts, and Twitter tweets—into advertisements targeted specifically at each of us.

Other modern-day applications that use parallel computing include:

- **Computational mathematics**: numerical linear algebra, numerical solution of ordinary and partial differential equations, optimization, combinatorics, graph theory, etc.
- **Scientific computation**: weather prediction, hurricane forecasting, climate modeling, astrophysics (galaxy and star cluster simulation, black hole gravitational wave prediction), chemistry (molecular dynamics, *ab initio* molecular structure prediction), bioinformatics (DNA sequence matching, protein sequence matching, protein structure prediction, pharmaceutical drug design), geology (seismic data analysis, oil and mineral prospecting), etc.
- **Engineering computation**: computational fluid dynamics, simulated wind tunnels, finite element analysis, circuit verification, integrated circuit chip component and wiring placement, etc.
Chapter 1. The Parallel Landscape

- Computational finance: asset pricing, derivative pricing, market modeling, algorithmic trading, . . .
- Big data analytics (as already mentioned): data mining, web indexing, user characterization, targeted advertising, . . .
- Security and cryptography: password cracking, cipher attacks, bitcoin mining and transaction processing, . . .
- Entertainment: computer games, computer generated imagery (CGI) for visual effects, computer animated films, . . .
- Fringe: Mersenne prime searching, Search for Extraterrestrial Intelligence (SETI), computer chess, . . .
- And the list goes on.

It wasn't always this way. In the beginning, up through the 1980s, parallel computers were esoteric and expensive. Each parallel computer vendor had its own proprietary hardware architectures, its own proprietary parallel programming languages (often nonstandard extensions to existing languages like Fortran and Lisp), and its own proprietary parallel software libraries. For the most part, the only ones with budgets big enough to afford parallel computers were large industrial organizations and government funded research laboratories. Consequently, parallel computing was used mostly for scientific and engineering applications.

A paradigm shift in parallel computing took place near the end of the twentieth century. Setting the stage was the development during the 1980s and early 1990s of inexpensive PC hardware, cheap Ethernet local area network hardware, standard TCP/IP network protocols, and the free Linux operating system. A 1995 paper* brought all these together and described how to build “Beowulf,” an example of what is now called a cluster parallel computer, from standard, off-the-shelf hardware and software, for a mere fraction of the cost of a proprietary supercomputer. This publication ushered in an era of parallel computing done with commodity chips and hardware rather than proprietary machines.

In addition, parallel programming shifted to using standard languages like Fortran, C, and later C++ with standard parallel programming libraries. Message Passing Interface (MPI), introduced in 1994, became the de facto standard for parallel programming on cluster parallel computers. OpenMP, introduced in 1997, became the de facto standard for parallel programming on multicore parallel computers.

However, at that time multicore machines were still not common; if you wanted lots of cores, you usually had to build a cluster of single-core ma-

chines. Consequently, parallel computing was still concentrated in industrial and government research settings and was still dominated by scientific and engineering applications.

A second paradigm shift in parallel computing took place in 2004. Until then, processor chip manufacturers had exploited Moore’s Law to steadily increase both the number of transistors on a chip and the chip speed, doubling the clock frequency about every two years. But by 2004, clock frequencies had gotten fast enough—around 3 GHz—that any further increases would have caused the chips to melt from the heat they generated (unless outfitted with cooling systems like Indianapolis 500 race cars). So while the manufacturers continued to increase the number of transistors per chip, they no longer increased the clock frequencies. Instead, they started putting multiple processor cores on the chip. A chip with two cores operating at 3 GHz is, theoretically, equivalent to a chip with one core operating at 6 GHz. The number of cores per chip continued to increase, until in 2014 as I am writing, everyone’s computer is a multicore parallel machine with two, four, eight, or more cores.

At the same time, memory chip densities continued to increase, until now everyone’s computer has 4 GB, 8 GB, 16 GB, or more of main memory. Furthermore, with the rise in popularity of computer games, both on desktop PCs and on gaming consoles, everyone’s computer has a graphics processing unit (GPU) with dozens or hundreds of cores. While originally intended for real-time 3-D video rendering, GPUs can do general-purpose calculations as well. Today’s PC is a powerful parallel computer with capabilities undreamed of in the 1980s, with computing power equivalent to a 1990s-era cluster requiring a whole rack of equipment.

The modern era of ubiquitous multicore machines opened parallel computing to a much broader range of applications than just scientific and engineering computation, as mentioned previously. Many of these newer applications eschew Fortran and C and MPI and OpenMP, which arose during the era when scientific and engineering applications dominated parallel computing. Modern applications also use newer languages like Java and newer programming paradigms like map-reduce. A prime example is Apache’s Hadoop, a map-reduce library written in Java, designed for parallel computation on massive data sets.

To use your modern computer—your modern parallel computer—to its full potential, it’s not enough to write plain old programs. You have to write parallel programs. These parallel programs have to exploit all the cores in the machine. If possible, these parallel programs also ought to utilize all the cores in the GPU. This book, BIG CPU, BIG DATA, is intended to teach you how to write parallel programs for multiple-core, GPU-equipped parallel computers.

Despite the vast increase in the PC’s computing power, there still are computational problems too big for a single node to handle. The problems
require more memory than can fit in a single node, or the problems require so much computation that it would take too long for them to finish when run on a single node (even a multicore node), or both. Such problems need to run on a *cluster* parallel computer, one with multiple nodes.

An individual user could afford to set up a small-scale cluster with perhaps two or three or four PC-class nodes. A small company or academic department could afford to set up a medium-scale cluster with perhaps one or two dozen nodes. A large-scale cluster with hundreds or thousands of nodes requires the budgetary resources of a large company or a government. The United States government, for example, funds a number of supercomputer centers at its national laboratories. These supercomputers are available to researchers who have government funding. The most powerful supercomputer in the U.S., according to the November 2014 Top500 List of the world’s fastest computers (www.top500.org), is the Titan computer at the Oak Ridge National Laboratory in Tennessee. Titan has 35,040 nodes, each with 16 CPU cores and 2,688 GPU cores, for a total of 560,640 CPU cores and 94,187,520 GPU cores. Titan is able to execute the Linpack linear algebra parallel processing benchmark at a rate of 17.6 petaflops ($17.6 \times 10^{15}$ floating point operations per second). Yet even Titan is only number two on the November 2014 Top500 List. First place goes to the Tianhe-2 computer at the National University of Defense Technology in Changsha, China. Tianhe-2 has 16,000 nodes and 3,120,000 CPU cores total, and it executes Linpack at 33.9 petaflops.

By the way, there’s nothing special about supercomputer hardware. Supercomputers nowadays use the same commodity chips as desktop PCs. Tianhe-2 uses Intel Xeon E5 CPU chips (2.2 GHz clock) and Intel Xeon Phi manycore accelerators. Titan uses AMD Opteron CPU chips (2.2 GHz clock) along with Nvidia Tesla K20x GPU accelerators. Supercomputers get their massive computational power by having large numbers of nodes and cores, not from superior chips or enormous clock frequencies.

What about the poor individual or department who needs large-scale computing power but doesn’t have large-scale funding? *Cloud computing* is an interesting and viable alternative. A cloud computing service, such as Amazon’s EC2 (aws.amazon.com/ec2/), will rent you the use of as many compute nodes as you want, for as long as you want, and charge you only for the actual CPU time you use. Better still, you don’t have to buy and maintain the computers, air conditioners, uninterruptible power supplies, network hardware, racks, cables, and so on needed to run a cluster parallel computer. The cloud service provider takes care of all that. Best of all, you don’t need a difficult-to-get government research grant. The cloud service provider is happy to charge your credit card.

It’s important to know how to write parallel programs for cluster parallel computers, to solve problems that are just too large for a single node. *BIG*
CPU, BIG DATA is intended to teach you how to write parallel programs for multiple-node cluster parallel computers, including clusters in the cloud.

To make sense of the bewilderingly diverse landscape that is modern parallel computing, I’m going to characterize parallel computing hardware, software, and applications along several dimensions. Figure 1.1 shows the eight dimensions. The next few sections discuss each dimension. Any particular parallel computer, parallel program, or parallel application can be pinpointed along each dimension, illuminating its place in the overall scheme of things.

**Hardware Dimensions**

<table>
<thead>
<tr>
<th>Single node</th>
<th>Multinode</th>
</tr>
</thead>
</table>

A node refers to an independent computer with its own CPU chip or chips, its own main memory, and its own network interface. A parallel computer can consist of a single node, or of multiple nodes. A multinode parallel computer is called a cluster.

<table>
<thead>
<tr>
<th>Single core</th>
<th>Multicore</th>
<th>Hyperthreaded</th>
</tr>
</thead>
</table>

A core refers to the hardware on a node that executes a stream of machine instructions. A node can have a single core, or multiple cores. Each core can also be hyperthreaded, meaning that the core can execute more than one stream of machine instructions.

<table>
<thead>
<tr>
<th>Unaccelerated</th>
<th>GPU</th>
<th>Manycore</th>
<th>FPGA</th>
<th>ASIC</th>
</tr>
</thead>
</table>

In addition to the CPU, a node can have zero or more accelerators. These are separate processors that can perform computations alongside the CPU. There are several kinds:

- A graphics processing unit (GPU) accelerator typically has a large number of cores that are mainly intended for graphics rendering but that can do general calculations.
- A manycore accelerator is similar to a general purpose CPU, except it typically has more cores than a CPU, and it omits all the peripheral interface circuitry (disk, network, display, USB, and so on) not needed for doing pure calculation.
- A field programmable gate array (FPGA) accelerator is a digital chip whose logic circuitry can be reconfigured during operation, letting you create customized high-speed processors.
- An application specific integrated circuit (ASIC) accelerator uses specialized chips hardwired to perform specific computations at very high speed. One example is bitcoin mining (bitcoin.org); you can buy ASICs that compute the SHA-256 cryptographic hash function, part of the Bitcoin protocol, at very high speed.
Having examined each hardware dimension by itself, let’s look at examples of parallel computers and see where they fall along the dimensions.

**Single-Core Computer**

Most computers back in the twentieth century were single node single core unaccelerated (Figure 1.2). In such a computer, the core has an *instruction unit* that reads the program’s machine instructions from memory, decodes them, and executes them; a number of *functional units*, such as adders, shifters, multipliers, and so on that carry out the machine instructions; and a number of high-speed *registers* that hold intermediate results. With one instruction unit, the core can execute only one stream of instructions—one thread—at a time. To execute more than one thread, the computer’s operating system must do a *context switch* from one thread to another every so often.

The computer has a *main memory* that stores program instructions and data. Because the main memory’s circuitry is much slower than the core’s
circuitry, a *level 2 (L2) cache* sits between the core and the main memory. The L2 cache is faster than the main memory, but smaller (typically a few megabytes). When the core reads a memory location into the instruction unit or into a register, an entire *cache line*, typically 64 or 128 bytes, is fetched from main memory and stored in the L2 cache; from there, the requested data goes into the core. If the core reads an adjacent memory location, the data is already in the L2 cache (*a cache hit*) and can be read quickly without waiting for the slow main memory.

Still, the L2 cache’s circuitry is not as fast as the core’s circuitry; so another cache, the *level 1 (L1) cache*, sits between the core and the L2 cache. The L1 cache is nearly as fast as the core but is smaller than the L2 cache. On its way to the core, data read from main memory is cached in L2 and in L1. If the instructions and data the program is accessing can fit entirely in L2, or better still L1, the core can run at nearly full speed. Otherwise, the core will have to pause while new data is fetched from the slow main memory, reducing the processing speed.

So as to keep the core running as fast as possible, the L1 and L2 caches are made as large as possible while still fitting on the CPU chip along with the rest of the core circuitry. Often, the majority of a CPU chip’s area is devoted to the caches.

**Multicore Computer**

<table>
<thead>
<tr>
<th>Single node</th>
<th>Multinode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core</td>
<td>Multicore</td>
</tr>
<tr>
<td>Unaccelerated</td>
<td>GPU</td>
</tr>
</tbody>
</table>

To increase the processing power while not increasing the clock frequency, computers switched from single core to *multicore* (Figure 1.3). Now there are two or more cores, each with its own instruction unit, functional units, registers, and L1 cache. The cores all share the L2 cache and the main memory. The operating system can run multiple threads simultaneously, one in each core, without needing to context switch. The threads are running truly in parallel. Theoretically, a *K*-core computer ought to run *K* times faster than a single-core computer. (This does not always happen in practice, though.)

**Hyperthreaded Computer**

<table>
<thead>
<tr>
<th>Single node</th>
<th>Multinode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core</td>
<td>Multicore</td>
</tr>
<tr>
<td>Unaccelerated</td>
<td>GPU</td>
</tr>
</tbody>
</table>
Replicating the instruction unit, functional units, registers, and L1 cache to get multiple cores requires a lot of transistors and chip area. To run more threads without needing quite so much area, multicore computers became hyperthreaded (Figure 1.4). In a hyperthreaded core, the instruction units are replicated, but not the functional units, registers, or L1 cache. A dual-hyperthreaded core can run two threads simultaneously without needing to context switch.
There’s a catch, though. As long as the two threads are using different functional units and registers, both threads can run at full speed. But if the threads both need to use the same functional unit or register at the same time, the hardware makes the threads take turns. While one thread is accessing the functional unit or register, the other thread stalls. This reduces the effective rate at which the threads can execute instructions. In my experience, a dual-hyperthreaded core typically does not run as fast as two regular cores; but it does run faster than one regular core.

Multicore Accelerated Computer

Lately, parallel computers are including accelerators alongside the multicore hyperthreaded CPUs. A GPU accelerator (Figure 1.5) repurposes a graphics card to do general calculations. A GPU card has numerous cores, anywhere from dozens to thousands of them, as well as its own main memory. The GPU’s main memory is linked with the CPU’s main memory over a high-speed bus, allowing data to be transferred between the CPU and GPU. Standard GPU programming libraries, such as Nvidia Corporation’s CUDA and the vendor-independent OpenCL, make it almost as easy to write GPU programs as it is to write CPU programs.

Both the CPU and the GPU have to deal with their main memory’s large latency (access time) relative to high speed of their cores. The CPU deals with it by interposing L1 and L2 caches between the cores and the memory, devoting the bulk of the chip area to cache, leaving room for only a few cores. The GPU deals with it by reducing or even eliminating the cache, devoting the bulk of the chip area to a large number of cores. The GPU then runs large numbers of threads simultaneously on its cores. When one thread stalls waiting to access main memory, another thread instantly takes its place. There are enough threads so that whenever a core goes idle, a thread is available that has completed its previous memory access and is ready to run again. In this way, the cores stay busy all the time. This technique is called latency hiding.

Depending on the nature of the problem, a GPU can perform calculations dozens or hundreds of times faster than a CPU. With GPUs incorporated into just about every modern computer, it’s important to know how to write GPU parallel programs, to take advantage of the GPU’s additional processing power. BIG CPU, BIG DATA is intended to teach you how to write parallel programs for GPU accelerated parallel computers. (I’m not going to cover the more esoteric manycore, FPGA, and ASIC accelerators in this book.)
Figure 1.4. Single node multicore hyperthreaded computer

Figure 1.5. Single node multicore hyperthreaded GPU accelerated computer
Single-Core Cluster Computer

<table>
<thead>
<tr>
<th>Single node</th>
<th>Multinode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core</td>
<td>Multicore</td>
</tr>
<tr>
<td>Unaccelerated</td>
<td>GPU</td>
</tr>
</tbody>
</table>

Turning away from single-node parallel computers, we come to multi-node parallel computers, or *clusters* (Figure 1.6). The cluster has some number of *backend nodes* that carry out computations. Each node has a single core plus its own main memory. We say the cluster has a *distributed memory*; the cluster’s memory is distributed across the nodes instead of concentrated in a single node. As we will see, the distributed memory has profound implications for the design of cluster parallel programs. The cluster has a dedicated high-speed *backend network* that allows the backend nodes to communicate with each other. The backend network may use commodity Ethernet hardware, or it may use specialized faster technology such as Infini-band, Myrinet, or Scalable Coherent Interface (SCI), or it may use a proprietary interconnect. The cluster usually also has a *frontend node*, connected to the Internet to let users log in and run parallel programs on the cluster, and connected to the backend network to control and monitor the backend nodes.

Multicore Cluster Computer

<table>
<thead>
<tr>
<th>Single node</th>
<th>Multinode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core</td>
<td>Multicore</td>
</tr>
<tr>
<td>Unaccelerated</td>
<td>GPU</td>
</tr>
</tbody>
</table>

However, nowadays it’s virtually impossible to buy a single-core node. So a modern cluster parallel computer consists of multiple multicore backend nodes (Figure 1.7). The cores might or might not be hyperthreaded.

Multicore Accelerated Cluster Computer

<table>
<thead>
<tr>
<th>Single node</th>
<th>Multinode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core</td>
<td>Multicore</td>
</tr>
<tr>
<td>Unaccelerated</td>
<td>GPU</td>
</tr>
</tbody>
</table>

In addition, the nodes of a modern cluster parallel computer might include accelerators (Figure 1.8). The Titan supercomputer looks like Figure 1.8; it has 35,040 nodes, each with 16 CPU cores and a GPU accelerator, and it uses a high-speed proprietary interconnect for its backend network.
Chapter 1. The Parallel Landscape

Figure 1.6. Multinode single core cluster computer

Figure 1.7. Multinode multicore cluster computer

Figure 1.8. Multinode multicore GPU accelerated cluster computer
Software Dimensions

A parallel program consists of multiple threads performing computations simultaneously. In a single-node multicore parallel computer, the threads run on the cores of the node. In a cluster parallel computer, the threads run on the cores of all the nodes.

<table>
<thead>
<tr>
<th>Uncoupled</th>
<th>Loosely coupled</th>
<th>Tightly coupled</th>
</tr>
</thead>
</table>

The computations performed by the threads can be *uncoupled*, *loosely coupled*, or *tightly coupled*. In an uncoupled computation, the threads do not communicate or coordinate with each other at all; each thread runs and produces its results independently of all the other threads. In a loosely coupled computation, the threads communicate with each other, but only infrequently; for example, the threads compute results independently of each other, but at the end of the program the threads communicate their individual results to each other and combine them into one overall result. In a tightly coupled computation, the threads communicate with each other frequently; for example, each thread executes a loop, and at the end of every loop iteration, the threads communicate the results of that iteration to the other threads before proceeding with the next iteration.

Coupling also refers to the quantity of data communicated between the threads. In an uncoupled computation, no data is exchanged between the threads. In a loosely coupled computation, a small amount of data is exchanged between the threads. In a tightly coupled computation, a large amount of data is exchanged between the threads. A particular parallel program might fall anywhere along the spectrum from uncoupled to tightly coupled.

<table>
<thead>
<tr>
<th>Non-shared memory</th>
<th>Shared memory</th>
</tr>
</thead>
</table>

In a *shared memory* parallel program, the data items the threads are accessing as they perform their computations—input parameters, intermediate values, output results—are stored in a single memory region that is shared by all the threads. Thus, any thread can get any other thread’s results simply by reading the appropriate locations in the shared memory. In a non-shared memory parallel program, the threads do not have access to a common shared memory.

<table>
<thead>
<tr>
<th>Non-distributed memory</th>
<th>Distributed memory</th>
</tr>
</thead>
</table>

In a *distributed memory* parallel program, the data items the threads are accessing are stored in multiple memory regions. Each thread can directly read and write locations in one of the memory regions, and the thread stores its own data in that memory region. Each thread can also access the contents of the other memory regions, but not by directly reading and writing them.
Chapter 1. The Parallel Landscape

Rather, one thread accesses another thread’s memory region via some form of explicit communication. The communication can take the form of message passing; the thread that owns the data sends a message that is received by the thread that needs to use the data. (MPI is a library of message passing subroutines of this sort.) Other possible communication mechanisms include remote procedure call (RPC), remote method invocation (RMI), and tuple space. In a non-distributed memory parallel program, the threads do not have any access to other threads’ memory regions.

Having examined each software dimension by itself, let’s look at examples of parallel computing software and see where they fall along all the dimensions.

**Multicore Parallel Program**

A parallel program intended to run on a single multicore node (including hyperthreaded cores) typically uses a shared memory model (Figure 1.9). The program runs in a single process with multiple threads, each thread executing on a different core. The program’s data is located in the computer’s memory. Because all the threads are part of the same process, and the process consists of a single address space, each thread can access all the program’s data; this is how the shared memory is achieved.

Typically, the data is partitioned into as many pieces as there are threads. Each thread computes and writes its own piece of the data. Each thread also reads the other pieces of the data as necessary. Thus, the threads communicate with each other by writing and reading the same shared memory locations. The threads can also coordinate with each other using synchronization primitives supported by most operating systems, such as semaphores, locks, and barriers.

Shared memory parallel programs can be tightly coupled, loosely coupled, or uncoupled. A loosely coupled or uncoupled program still looks like Figure 1.9; the only difference is the frequency with which one thread accesses another thread’s data, or the amount of data accessed.

Shared memory parallel programs are just multithreaded programs, no more and no less. You can write a shared memory parallel program using the
built-in threading constructs of a programming language or an operating system, such as Java threads or Unix pthreads. However, folks who need to do parallel computing often are not experts in writing threaded code. They want to write high-level application programs to solve their computational problems; they don’t want to have to write low-level code to create threads, acquire and release semaphores and locks, destroy threads, and so on. Consequently, most folks use a parallel programming library, or application programming interface (API), to write shared memory parallel programs. The API exposes high-level application-oriented parallel programming constructs to the programmers, and the API handles all the low-level threading details under the hood.

OpenMP (www.openmp.org) is a widely used API for shared memory parallel programming. First released in 1997, and now in its fourth revision (version 4.0 of the OpenMP specification was released in July 2013), OpenMP supports parallel programming in the Fortran, C, and C++ languages.

I prefer to program in Java. So I prefer not to use OpenMP, which does not—and, in my belief, never will—support Java. Instead, I’m going to use the Parallel Java 2 Library, which I have developed, to teach you shared memory parallel programming in Java.

Cluster Parallel Program

A parallel program intended to run on a cluster of single-core or multi-core nodes typically uses a distributed memory model (Figure 1.10). The program runs in multiple processes, one process for each core of each backend node. Each process has one thread running on the core plus data located in the node’s memory. Typically, the data is partitioned into as many pieces as there are threads. But because the threads and data pieces are in different processes with different address spaces, the memory is not shared. Each thread can access its own data directly. But if one thread needs to use a data item located in another thread’s memory region, message passing has to take place.

For example, suppose thread 7 needs a data item located in thread 3’s memory region. Thread 3 has to retrieve the data item from its memory and load the data into a message of some kind. Because the threads are running
Figure 1.9. Shared memory parallel program running on a multicore node

Figure 1.10. Distributed memory parallel program running on a cluster of multicore nodes
on different nodes, thread 3 must send the message over the cluster’s back-end network to thread 7—an \textit{inter-node} message. Thread 7 has to receive the message and extract the data. Or suppose thread 6 needs a data item located in thread 4’s memory region. Although the threads are running on the same node, because the threads are running in different processes (different address spaces), a message still has to go from thread 4 to thread 6—an \textit{intra-node} message.

Message passing can, of course, be more complicated than these simple examples. One or more owner threads might need to send one or more data items to one or more recipient threads. The threads’ programs need to be coded to invoke message passing operations explicitly; this increases the complexity and programming effort for cluster parallel programs.

Cluster parallel programs can be tightly coupled, loosely coupled, or uncoupled. An uncoupled program still looks like Figure 1.10, except there is no message passing. A loosely coupled program looks like Figure 1.10, but does fewer or less frequent message passing operations, or sends less data, than a tightly coupled program.

You can write a cluster parallel program using the \textit{interprocess communication (IPC)} constructs of an operating system or using networking software like TCP sockets. However, folks who need to do parallel computing often are not experts in writing IPC or networking code. They want to write high-level application programs to solve their computational problems; they don’t want to have to write low-level code to open and close sockets, format data into and out of messages using some protocol, and so on. Consequently, as with shared memory parallel programming, most folks use a parallel programming library or API to write cluster parallel programs. The API exposes high-level application-oriented parallel programming constructs to the programmers, and handles all the low-level networking details under the hood.

To achieve acceptable performance, a tightly coupled cluster parallel program needs to use a fast, low-overhead message passing library. \textit{MPI} (www.mcs.anl.gov/research/projects/mpi/) is a widely used API for cluster parallel programming. First released in 1994, and updated to Version 2 in 1997 (Version 3 is in the works), MPI supports parallel programming in the Fortran, C, and C++ languages. MPI is a library of message passing subroutines; programs written to call these subroutines can run on any machine that has an MPI library installed. Often, a platform-specific MPI implementation is used to wring the fastest possible performance out of the hardware. Platform-independent MPI implementations are also available but tend to be slower.

Because I prefer to program in Java, I prefer not to use MPI, which does not—and, in my belief, never will—support Java. Also, I’m not fond of MPI’s huge and complicated API. The Parallel Java 2 Library includes message passing capabilities via a simple API called \textit{tuple space}. However, the
Parallel Java 2 Library is intended mainly for uncoupled and loosely coupled cluster parallel programs. While tightly coupled cluster parallel programs can be written with Parallel Java 2, the tuple space’s platform independent implementation is not designed to achieve the highest possible speed. If you need extremely fast message passing, use MPI.

**Multicore Cluster Parallel Program**

<table>
<thead>
<tr>
<th>Single node</th>
<th>Multinode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core</td>
<td>Multicore</td>
</tr>
<tr>
<td>Unaccelerated</td>
<td>GPU</td>
</tr>
<tr>
<td>Uncoupled</td>
<td>Loosely Coupled</td>
</tr>
<tr>
<td>Non-shared memory</td>
<td>Shared memory</td>
</tr>
<tr>
<td>Non-distributed memory</td>
<td>Distributed memory</td>
</tr>
</tbody>
</table>

A tightly coupled parallel program running on a cluster of multicore nodes requires frequent exchange of copious data both between processes running on the same node and between processes running on different nodes, as shown in Figure 1.10. But it doesn’t make sense to do message passing between processes running on the same node. Sending data between processes on a node is typically much slower than accessing the data directly. What does make sense is to use the shared memory model within each node and the distributed memory model between nodes—a hybrid shared/distributed memory model (Figure 1.11). On each node there is one process with multiple threads, one thread per core, with the threads directly accessing each other’s data in shared memory. When data has to go from one process (node) to another, message passing is used. By eliminating the bulk of the messages needed in the pure distributed memory model (Figure 1.10), the program’s performance is improved.

Some parallel programs have more than one level of parallelism. A program might perform many separate, uncoupled computations, which can therefore be done in parallel. Each of these might itself be a tightly coupled parallel computation. Such a program is ideally suited to run on a multicore cluster parallel computer using the hybrid shared/distributed memory model. The computations run in separate parallel processes on separate nodes, with no message passing between computations. Each computation runs in multiple parallel threads on separate cores in the same node, all the threads accessing the computation’s data in shared memory.
Figure 1.11. Hybrid shared/distributed memory parallel program running on a cluster of multicore nodes
Chapter 1. The Parallel Landscape

GPU Accelerated Parallel Program

All the previous parallel software options involved unaccelerated nodes. Figure 1.12 depicts a parallel program that uses a GPU accelerator. The figure shows the simplest kind of GPU parallel program: running in a single CPU thread, on a single CPU core, with all the parallelism in the GPU. The red arrows show what I like to call the GPU computational arc:

- The CPU sets up input data for the computation in the CPU memory. Often the data is an array or matrix consisting of many, many elements.
- The CPU sends the input data from the CPU memory to the GPU memory.
- The CPU launches a large number of GPU threads. Each thread will execute a kernel function (denoted by “K” in Figure 1.12). The whole assemblage of GPU threads executing kernel functions is called the computational kernel, or just kernel. The CPU waits for the kernel to finish.
- Each GPU thread executes the kernel function on a GPU core. Often, each individual data element is computed by its own separate GPU thread. The computation’s output data is stored back in the GPU memory.
- When the kernel finishes, the CPU wakes up and sucks the output data from the GPU memory back to the CPU memory.
- The CPU outputs the computation’s results.

The GPU cores achieve their best performance when they all execute the exact same stream of machine instructions in lockstep, each on different data items—what is called single instruction stream multiple data stream (SIMD) parallelism. The GPU cores also achieve their best performance when the data they are accessing is stored in a regular pattern in memory, such as array or matrix elements in contiguous memory locations. A program that has a lot of data-dependent branching, with different instruction sequences being executed depending on the data values, or a program that has irregular data access patterns, such as pointer chasing through linked data structures, will not perform well on a GPU. Thus, typically only a portion of a GPU parallel program runs on the actual GPU—namely, the SIMD, regular-data-access, com-
A GPU accelerated parallel program might run on more than one CPU core: the computational kernel runs in parallel on the GPU cores, and the non-kernel portion runs in parallel on the CPU cores. The CPU threads might run in parallel with the GPU threads, rather than waiting for the kernel to finish. Multiple kernels might run on the GPU at the same time. And with a GPU accelerated cluster, all this could be happening in parallel repeatedly on multiple nodes. The possibilities for parallelism are endless.

Nvidia Corporation pioneered general purpose computing on GPUs with their proprietary *Compute Unified Device Architecture (CUDA)* and the programming API that goes with it. CUDA supports writing GPU kernel functions in Fortran, C, and C++. The CPU main programs are written in the same languages. *OpenCL* (www.khronos.org/opencl) is a more recent, vendor neutral API for GPU programming. First released in 2009, and last updated in March 2014, OpenCL uses its own programming language based on C.

The Parallel Java 2 Library supports GPU parallel programming via a combination of CUDA and Java. The GPU kernel functions are written in C...
or C++ using CUDA. (OpenCL support is a planned enhancement.) The main programs running on the CPU are written in Java, using classes that provide high level abstractions of the GPU. Under the hood, these classes access the GPU via Java’s native interface capability. (At this time, I don’t know of a way to write GPU kernel functions directly in Java. A Java compiler targeting GPUs would make for a very interesting project!)

**Application Dimensions**

Parallel computing applications are characterized along two orthogonal dimensions: little CPU—big CPU, and little data—big data.

**Little CPU Little Data Application**

- Little CPU
- Medium CPU
- Big CPU
- Little Data
- Medium Data
- Big Data

A little CPU little data application works with only a small amount of data, and does only a few calculations (CPU cycles) with each data item. Still, the calculations are or can be done in parallel. A spreadsheet is an example. Compared to a supercomputer program, a spreadsheet works with very little data (the cell values) and does very little computation (the cell formulas). However, the cells can be calculated in parallel, as long as any data dependencies between cells are obeyed.

**Big CPU Little Data Application**

- Little CPU
- Medium CPU
- Big CPU
- Little Data
- Medium Data
- Big Data

A big CPU little data application also works with only a small amount of data, but it spends a large amount of CPU time doing calculations with that data. Doing the calculations in parallel can speed up the application. Cryptographic applications are often of this kind. Bitcoin mining is one example. A bitcoin “block” is a piece of digital cash. It occupies just a few kilobytes of data. But determining the value of a certain field of the block—so-called “mining” the bitcoin—requires calculating the SHA-256 cryptographic hash function many, many times. These calculations can be, and usually are, performed in parallel. Also, bitcoin miners can mine multiple blocks in parallel.

**Little CPU Big Data Application**

- Little CPU
- Medium CPU
- Big CPU
- Little Data
- Medium Data
- Big Data
A little CPU big data application devotes only a little bit of CPU time to each data item, but works with an enormous number of data items. Consequently the application can take a long time to run, and processing the data in parallel can speed it up. Map-reduce, pioneered by Google and implemented in the popular Apache Hadoop, is a widely used paradigm for parallel big data applications. Apache’s “Powered by Hadoop” web page* shows that many major Internet players—Amazon, eBay, Facebook, Hulu, LinkedIn, Spotify, Twitter, Yahoo, and others—use Hadoop running on multicore clusters for their big data analytics. Google also does big data analytics with their own map-reduce software. The Parallel Java 2 Library includes Parallel Java Map Reduce (PJMR), a lightweight map-reduce framework built on top of the Library’s cluster parallel programming capability.

### Big CPU Big Data Application

<table>
<thead>
<tr>
<th>Little CPU</th>
<th>Medium CPU</th>
<th>Big CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little Data</td>
<td>Medium Data</td>
<td>Big Data</td>
</tr>
</tbody>
</table>

Finally, a big CPU big data application works with lots and lots of data and does lots and lots of calculations with each data item. Scientific and engineering calculations on supercomputers are of this kind. As an example of the extreme scale of these applications, consider the LAMMPS molecular dynamics program, which simulates the motion of atoms from first principles of physics, running on the Keeneland supercomputer at the Oak Ridge National Laboratory. Keeneland is a medium-size cluster of 120 nodes, with two CPU cores and three GPU accelerators per node. LAMMPS running a one billion atom benchmark for 100 time steps requires about half a terabyte \( (5 \times 10^{11}) \) bytes of data and would take the better part of an hour (2,350 seconds) on one core of Keeneland. Running on the entire cluster, the same benchmark takes just 17.7 seconds.†

### Points to Remember

- Parallel computing hardware can be characterized along three dimensions: single node—multinode, single core—multicore—hyperthreaded, and unaccelerated—accelerated.
- Parallel computing software can be characterized along three dimensions: uncoupled—loosely coupled—tightly coupled, non-shared memory—shared memory, and non-distributed memory—distributed memory.
- Parallel computing applications can be characterized along two dimensions: little CPU—big CPU, and little data—big data.

---

* [http://wiki.apache.org/hadoop/PoweredBy](http://wiki.apache.org/hadoop/PoweredBy)
• It’s important to know how to write parallel programs for multicore computers, clusters, and GPUs.