## Chapter 3

Parallel Software

- **Part I. Preliminaries**
  - Chapter 1. What Is Parallel Computing?
  - Chapter 2. Parallel Hardware
  - **Chapter 3. Parallel Software**
  - Chapter 4. Parallel Applications
  - Chapter 5. Supercomputers
- **Part II. Tightly Coupled Multicore**
- **Part III. Loosely Coupled Cluster**
- **Part IV. GPU Acceleration**
- **Part V. Map-Reduce**
- **Appendices**
Parallel computing software is characterized along three dimensions (Figure 3.1): coupling, shared memory, and distributed memory. Different parallel programs fall at different locations along these three 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.

| Uncoupled | Loosely coupled | Tightly coupled |

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.

| Non-shared memory | Shared memory |

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.

| Non-distributed memory | Distributed memory |

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. 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 3.2). 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 3.2; 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 4.5 of the OpenMP specification was released in November 2015), OpenMP supports parallel programming in Fortran, C, and C++.

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 3.3). 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 on different nodes, thread 3 must send the message over the cluster’s back-end network to thread 7—an 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 intra-node message. The threads’ programs need to be coded to invoke message passing operations explicitly; this increases the complexity and programming
**Figure 3.2.** Shared memory parallel program running on a multicore node

**Figure 3.3.** Distributed memory parallel program running on a cluster of multicore nodes
effort for cluster parallel programs.

Message passing can, of course, be more complicated than these simple examples. One owner thread might need to send data items to several recipient threads. One recipient thread might need to gather data items from several owner threads. Every thread might need to send data items to and receive data items from every other thread.

Cluster parallel programs can be tightly coupled, loosely coupled, or uncoupled. An uncoupled program still looks like Figure 3.3, except there is no message passing. A loosely coupled program looks like Figure 3.3, 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 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 and messaging protocol details under the hood.

To achieve acceptable performance, a tightly coupled cluster parallel program needs to use a fast, low-overhead message passing library. Message Passing Interface (MPI) (mpi-forum.org) is a widely used API for cluster parallel programming. First released in 1994, updated to Version 3.0 in September 2012, with Version 4.0 in the works, MPI supports parallel programming in Fortran, C, and C++. 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. (It fills an 852-page book!) The Parallel Java 2 Library includes message passing capabilities via a simple API called 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.
Chapter 3. Parallel Software

Multicore Cluster Parallel Program

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 3.3. 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 3.4). 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 an-
other, message passing is used. By eliminating the bulk of the messages needed in the pure distributed memory model (Figure 3.3), 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.

**GPU Accelerated Parallel Program**

All the previous parallel software options involved unaccelerated nodes. Figure 3.5 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 3.5). 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, computational kernel. The rest of the parallel program runs on the CPU.

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 May 2017, OpenCL uses its own programming language based on C and C++. 

Figure 3.5. GPU accelerated parallel program
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!)

Points to Remember

- 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.
- A parallel program running on a single multicore node consists of one process with multiple threads, one thread for each core on the node.
- All the threads can access the program’s data stored in the process’s address space (shared memory).
- A parallel program running on a cluster of multicore nodes can consist of one process for each core on each node, with one thread in each process.
- The program’s data is partitioned and distributed among the processes (distributed memory).
- Each process accesses its own data directly. Each process accesses other processes’ data via message passing.
- Alternatively, a parallel program running on a cluster of multicore nodes can consist of one process for each node, with one thread in the process for each core on the node.
- The program’s data is partitioned and distributed among the processes. Within each process, the data is further partitioned among the threads (hybrid shared/distributed memory).
- Each thread accesses data in its own process directly. Each thread accesses data in other processes via message passing.
- A parallel program running on a GPU accelerated node can consist of one thread on the CPU and multiple threads on the GPU.
- The program executes the computational arc: the CPU uploads the input data to the GPU, the GPU threads compute the output data, the CPU downloads the output data from the GPU.