Chapter 3

HIGH PERFORMANCE COMPUTING

Since the advent of the first digital computer decades ago, prices of computers have dropped significantly, making personal computers affordable. Meanwhile, the performances [in terms of the memory size and speed of the central processing unit (CPU)] double every 18 to 24 months (the so-called Moore's law). Computers are usually connected to one another to form a web of computers called Internet. A conceivable avenue of achieving high-performance computing is to coordinate together the vast number of otherwise idle computers on the network to tackle single tough tasks of computation. This is the very idea behind grid computing where both data and computing power are shared and accessible to a user. We will demonstrate an implementation of the so-called distributed computing to boost the performance in this chapter. Before this, let’s introduce the other high performance computing via parallelism in Java.  

3.1 Parallel Computing

There arise cases where a task can be divided into independent pieces. If each piece is taken care of by an individual CPU and the multiple CPU’s are run concurrently in the system, then ideally we expect a time saving by a factor of the total number of CPU’s in the system. The actual saving depends on

---

1Language design features such as Java’s checks of array indexes and references to objects for out-of-bound and null-pointer exceptions at runtime make Java a secure and reliable programming platform. They however have detrimental effects on technical computing. The arrays of arrays structure for multidimensional arrays in Java further hurts its numerical performance. As compiler optimization technologies advance, Java code can achieve 50% to 90% performance of highly optimized Fortran.

Since the book focuses on numerical computation, we play down handy applications of Java’s container classes such as java.util.Vector. When collection classes are nevertheless used, we point out that the overhead due to extravagant object creation and type casting should be avoided.
the nature of the task and on the hardware architecture of the multi-processor system.

If, for example, data are shared among the processors, deliberate synchronization of the computing processes between data updates has to be devised. The issue of synchronization for jobs of a subtle nature like this is to be reminded of. Otherwise, the program ends up computing what is not meant to do because of corrupted data. We will see examples of synchronization in Chapter 13.

If the scattered computers are inter-connected via slow links, communications overhead counterplays gains in parallelization. The slower the link or the more frequent for the task to exchange data among computers, the severer is the penalty.

Bearing in mind the precautions, we show straightforward implementation of parallelism in Java, via the Thread class.

3.2 Java Threads

A thread is a separate executing process in a program. We have experienced, without notice, threads in Java's windowed programming in the previous chapter. Threading is in fact indispensable to interactivity in windowed programming. Consider the case where a lengthy job is running while the user clicks on a button on the menu of the window. Without threads, the application would not respond to the click until the lengthy job is finished. A window object is therefore a thread.

In a single CPU system, a thread is run by the CPU at one instance of time. It's suspended when, at the next instance of time, the CPU switches its attention to another thread. The switching over is usually so swift that the user does not notice the pause in the execution of individual threads. Examples are the playing of video/audio files while writing emails. In addition, consider that an application is faced with a slow process which can be due to hardware, such as reading from a slow device (tape) or network connection. It is then desirable for the application to spawn a separate thread for this slow process. The application can then impart time to other processes while waiting for the slow reading to finish its course. Chapter 13 shows one example of the so-called buffered I/O. Threading is thus handy for multi-process and/or multi-user applications. There, however, exist no gains in computation in such single CPU systems.

Once a class extends the Thread class (or implements the Runnable interface, a method called run() has to be provided. The method run() of the class then runs as a thread.

Dual-CPU PC's and quad-CPU servers are getting popularity due to their little extra cost. When they are administered by appropriate operating systems,