Parallelization and Load Balancing Techniques for HPC

Siddhartha Khaitan
Iowa State University, USA

James D. McCalley
Iowa State University, USA

INTRODUCTION

As multicore systems become ubiquitous in desktop, and in mobile and embedded systems, interest in high performance computing (HPC) techniques has increased. Further, several computation intensive tasks demand use of high performance computing resources (Raju et al., 2009, Pande et al., 2009, Varré et al., 2011, Gupta et al., 2008) since sequential computing platforms are proving to be incapable of fulfilling the computational demands in these domains. Hence, researchers are using parallelization techniques. However, parallelization also brings the need of achieving load-balancing, since an unbalanced load distribution is likely to lead to wastage of processors and increased total completion time.

In this chapter, we discuss three different scheduling techniques which are used for achieving load-balancing. These techniques are static scheduling, master-slave scheduling and work-stealing. To show a concrete example of parallelization approach, we show the example of multi-threading in Java (Arnold et al., 2000). We discuss the relative advantages and disadvantages of multi-threading implementation in Java.

BACKGROUND

Parallel Programming

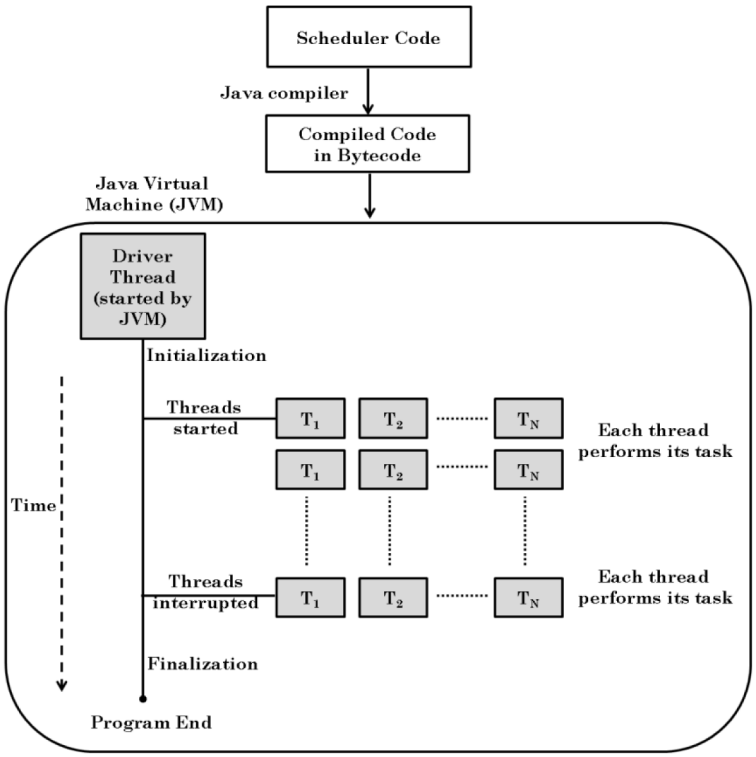
Parallel programming requires careful design to ensure functional correctness and avoid race conditions. Further, to gain performance, the data structure needs to be carefully designed to minimize locking and maximize independent progress of each process. This is because from Amdahl's law (Amdahl, 1967), the performance improvement achieved from parallelizing a program is limited by the fraction of time spent in the sequential part. In other words, Amdahl's law suggests that since the sequential parts of an algorithm are the slowest, these parts present a bottleneck in performance scaling. Thus, to achieve high performance, careful management of work-division among computing elements and aggregation of the result is required.

The parallelization approach can be classified based on whether they use hardware parallelism or software parallelism. Hardware parallelism refers to the kind of parallelism defined by the machine architecture and hardware multiplicity. Hardware parallelism indicates the peak performance of a processor. As an example, if a processor issues N instructions in each machine cycle, it is referred to as a N-issue processor. In contrast, software parallelism is defined by the control and data dependence of programs. Here the degree of parallelism is shown in the program flow path or program profile. The amount of software parallelism, which can be obtained, depends on the algorithm and programming style.

Recently, researchers have used different techniques such as multiprocessing and multithreading (Tullsen et al., 1995). Compared to processes, threads are lightweight objects and hence, switching between them incurs less overhead than switching between processes. Moreover, threads

DOI: 10.4018/978-1-4666-5202-6.ch159

Figure 1. Use of multi-threading for parallelization using Java



share memory and hence communication between them is also less expensive than that between the processes. Thus, using threads, the intermediate variables and the final result can be easily shared. Several programming languages inherently provide constructs to do parallel programming. We discuss this with the example of multi-threading in Java, as shown in Figure 1.

Figure 1 shows an example of how a scheduling technique is implemented in Java, assuming that the tasks have equal priorities. First, the Java compiler generates bytecodes which is an architecture-independent intermediate format. Bytecode enables transporting the code to multiple hardware and software platforms. These bytecodes are the instruction sets of the Java virtual machine. Thus, because of the interpreted nature of Java, the same Java language byte code can run on any platform and thus, the complexities of binary distribution and versioning are avoided.

In Java, multithreading is supported at the language level and the run-time system provides monitor and condition lock primitives. Depending on the requirements of the scheduler, JVM spawns multiple threads. Each thread is an object of Thread class in Java. Each object of the Thread class, encapsulates both the data and methods required for separate threads of execution. Different threads are executed using time-slicing and thus, the threads share the processor. In this manner, concurrency using multithreading can be achieved using even a single processor. In multicore platforms, different threads can be scheduled on different cores and thus, parallel resources can be explicitly used. Java's high-level system libraries are thread-safe and hence, multiple concurrent threads can access the functionality provided by the libraries without causing a conflict.

By leveraging the flexibility of Java implementation, the scheduler can be used in distributed environments. Further, due to the in-built security