Challenges and Gains

Another challenge of harnessing parallelism is the rising heterogeneity of multi-processor systems. CPU manufacturers take pride in delivering reasonably priced consumer-oriented systems with four or eight processing cores, and high-end server systems with dozens of cores. However, nowadays a mid-range workstation or a high-end laptop often comes equipped with a powerful graphics processing unit (GPU), with support for *hundreds* of concurrent threads. As if the two kinds of parallelism were not enough, Infrastructure-as-a-Service (IaaS) price drops sprout weekly, making thousand-core clouds accessible in a blink of an eye.

Note Herb Sutter gives an excellent overview of the heterogeneous world that awaits parallelism frameworks in his article "Welcome to the Jungle" (2011). In another article from 2005, "The Free Lunch Is Over," he shaped the resurgence of interest in concurrency and parallelism frameworks in everyday programming. If you should find yourself hungry for more information on parallel programming than this single chapter can provide, we can recommend the following excellent books on the subject of parallel programming in general and .NET parallelism frameworks in particular: Joe Duffy, "Concurrent Programming on Windows" (Addison-Wesley, 2008); Joseph Albahari, "Threading in C#" (online, 2011). To understand in more detail the operating system's inner workings around thread scheduling and synchronization mechanisms, Mark Russinovich's, David Solomon's, and Alex Ionescu's "Windows Internals, 5th Edition" (Microsoft Press, 2009) is an excellent text. Finally, the MSDN is a good source of information on the APIs we will see in this chapter, such as the Task Parallel Library.

The performance gains from parallelism are not to be dismissed lightly. I/O-bound applications can benefit greatly from offloading I/O to separate thread, performing asynchronous I/O to provide higher responsiveness, and scaling by issuing multiple I/O operations. CPU-bound applications with algorithms that yield themselves to parallelization can scale by an order of magnitude on typical consumer hardware by utilizing all available CPU cores or by two orders of magnitude by utilizing all available GPU cores. Later in this chapter you will see how a simple algorithm that performs matrix multiplication is sped up 130-fold by only changing a few lines of code to run on the GPU.

As always, the road to parallelism is laden with pitfalls—deadlocks, race conditions, starvation, and memory corruptions await at every step. Recent parallelism frameworks, including the Task Parallel Library (.NET 4.0) and C++ AMP that we will be using in this chapter, aim to reduce somewhat the complexity of writing parallel applications and harvesting the ripe performance profits.

Why Concurrency and Parallelism?

There are many reasons to introduce multiple threads of control into your applications. This book is dedicated to improving application performance, and indeed most reasons for concurrency and parallelism lie in the performance realm. Here are some examples:

- Issuing asynchronous I/O operations can improve application responsiveness. Most GUI applications have a single thread of control responsible for all UI updates; this thread must never be occupied for a long period of time, lest the UI become unresponsive to user actions.
- Parallelizing work across multiple threads can drive better utilization of system resources. Modern systems, equipped with multiple CPU cores and even more GPU cores can gain an order-of-magnitude increase in performance through parallelization of simple CPU-bound algorithms.
- Performing several I/O operations at once (for example, retrieving prices from multiple travel websites simultaneously, or updating files in several distributed Web repositories) can help drive better overall throughput, because most of the time is spent waiting for I/O operations to complete, and can be used to issue additional I/O operations or perform result processing on operations that have already completed.