White Paper

How to sound like a Parallel Programming Expert
Part 1: introducing concurrency and parallelism

Tim Mattson, Principal Engineer, CTG, Intel Corp

Introduction

Parallel computing has arrived. I am writing this on a dual core laptop. My son just bought a game console with 9 cores. My cell phone has four processors. Graphics processing units (GPUs) have dozens of cores. And the world's fastest supercomputers have over 100 thousand cores.

Parallel computing is ubiquitous. Why? Parallelism has been the path to ultimate performance for over two decades. But the push into every segment of the computing market isn't driven solely by the need for speed. The issue is energy efficiency and getting the maximum performance per watt of power consumed. The energy consumed by a chip increases steeply as you increase the frequency. At some point, you just can't push the frequency on a chip without putting more power into a socket than you can utilize efficiently, and properly dissipate. The only solution is to keep the clock frequency for an individual processor fixed and deliver increasing performance by increasing the number of cores within that processor

And that is exactly what the industry has done; the result being that everywhere you look, you will find parallel computers. The problem is that to take advantage of these parallel systems, you need multiple streams of instructions to pass through the computational elements at one time. And that may require significant work to recast your problem and the supporting software so it can generate multiple streams of instructions; a type of programming called "parallel programming".

Mastering the art of parallel programming is difficult. In the past, parallel programming experts were only needed for high-end supercomputing[i]. Today, however, the entire software industry needs to make a wholesale shift to parallel programming. While becoming a parallel programming expert is difficult and takes years of hard work, most people only need to understand the issues behind parallelism at a high level. For most people, all they need is to know how to "talk like a parallel programming expert".

That is the goal of this series of brief papers. We will provide the information you need to correctly use and understand the jargon that has sprung up around parallel computing. We will do this in four parts

- Part 1: introduction plus the difference between concurrency and parallelism
- Part 2: Parallel hardware
- Part 3: Issues in parallel programming
- Part 4: Parallel programming made easy

If you stick with me and make it through all four parts, you'll understand parallel computing at a high level, know how to correctly use the jargon of parallel programming, and just maybe, you'll get hooked and decide to not just "talk like" an expert; you might decide to "become" an expert.

Concurrency

The fundamental concept behind parallel computing is concurrency.

> Concurrency: A property of a system in which multiple tasks that comprise the system remain active and make progress at the same time.

Concurrency is an old concept. All mainstream operating systems (OS) in use today make heavy use of concurrency. They keep multiple threads of execution active and quickly swap between them. This way, if the system is stalled waiting for something, other work can make progress. Since the clock on a computer measures time in nanoseconds while a human "clock-tick" is measured in milliseconds, a concurrent OS can create the impression that multiple tasks are running at the same time, even when the system has only a single processing element.

Writing concurrent software can be difficult. First, a programmer must identify the concurrency in their problem by finding chunks of work that can run at the same time. The program must manage how data is shared between tasks and take steps to make sure that regardless of how the execution of different tasks is interleaved, correct results are produced. And finally, some sort of notation or Application Programming Interface (API) must be used to express the concurrency in the program's source code.

Parallelism

What is parallelism?

> Parallelism: Exploiting concurrency in a program with the goal of solving a problem in less time.

Parallel computing is an application of concurrency. You can have no parallelism without concurrency. And to execute a parallel program in parallel, you must have hardware with multiple processing elements so concurrent tasks execute in parallel.

For example, ray tracing is a common approach for rendering images. The problem naturally contains a great deal of concurrency since in principle, each ray of light can be handled as an independent task. We can queue these tasks up for execution and use a pool of threads to run them in parallel on a parallel computer. In other words, we exploit the concurrency in ray tracing to create a parallel program to render a fixed sized image in less time.

Algorithms

An algorithm is a sequence of steps designed to solve a problem. For traditional serial algorithms, the steps run one at a time in a well defined order. Not surprisingly, when concurrent and parallel algorithms are considered, things get a bit more complicated.

Concurrency in an algorithm implies that instead of a single sequence of steps, you have multiple sequences of steps that execute together. These steps are interleaved in different ways depending on how the tasks are scheduled for execution. This means the order of memory access operations will vary between runs of a program. If those memory access operations mix reads and writes to the same location, results can vary from one run of a program to the next. This is called a race condition or just a "race" for short. Hence, when designing algorithms that include concurrency, you have to use constructs that force ordered access to memory in just those places where read/write sharing takes place. Getting these so-called synchronization constructs in just the right places can be quite challenging.

We will consider two cases where we use concurrency in an algorithm. When the problem fundamentally requires concurrency just to accomplish the assigned task, we call it a concurrent algorithm. For example, consider a network attached printer with a print queue to manage jobs across many users. This print queue is fundamentally concurrent just to accomplish its job.

The other case where we work with concurrency within an algorithm is a parallel algorithm. Based on the definition of parallelism we discussed above, you can probably guess what we mean. This is an algorithm designed to take an operation and work concurrency into it so the problem is solved in less time. These parallel algorithms will be the focus of our discussions on learning how to sound like a parallel programming expert.

The study of parallel algorithms can be overwhelming. There are thousands of seemingly different algorithms in the literature of parallel programming. Fortunately, this is a mature field and we have made great progress at organizing them into a straightforward pedagogically-useful system. A particularly interesting approach uses a Design Pattern Language to capture the knowledge taken for granted by expert parallel algorithm designers and put it into a form people can easily absorb. As shown in the book "Patterns for Parallel Programming" (Mattson, Sanders, and Massingill, 2004) parallel algorithms fall into three broad categories based on the primary source of concurrency in the design:

- Task-based algorithms
- Data parallel algorithms
- The flow of data through a network of processing stages

These algorithm strategy patterns then map onto lower level implementation-oriented patterns that talk about how to express the parallel algorithms using different parallel programming environments.

We are getting ahead of ourselves, though. We have a great deal to learn about parallel hardware, parallel performance issues, and correctness before we can return to how we actually create parallel software.

Next steps

We have described the overall parallel programming problem and have explained the ideas of concurrency and parallelism. We closed with a brief hint at how to think about parallel algorithms. Next we will consider the parallel hardware required to run a parallel program.

About the Author

Dr. Timothy Mattson
Tim Mattson is a Principle Engineer at Intel working in the Microprocessor Technology laboratory. He is also a parallel programmer. Over the last 20 years, he has used parallel computers to make chemicals react, shake up proteins, find oil, understand genes, and solve many other scientific problems. Tim's long term research goal is to make sequential software rare. This means he has many years of hard work ahead of him.

[i] http://www.top500.org/