How to sound like a Parallel Programming Expert
Part 3: parallel computing issues
Tim Mattson, Principal Engineer, CTG, Intel Corp

Introduction

You run a program on a parallel computer so that it will run faster. Or you use parallel programming so you can run a larger problem without increasing the time. In both cases, the issue comes down to performance. Parallel programming is all about performance.

At the same time, it isn’t useful if you build a program that produces incorrect results in less time. There are errors unique to parallel programs that you need to understand if you want to sound like an expert when discussing parallel computing.

This is the goal of this short note: to understand the terminology used when talking about performance and correctness.
Performance on Parallel computers

Performance comes in two flavors in computing. We talk about the “latency” of a task as the elapsed time to complete the single task. Latency is critical in interactive computing applications. If you move a joy stick during a computer game, you want the resulting action to be nearly instantaneous with very little latency.

The alternative to latency when talking about performance is “throughput”. This refers to the aggregate performance of a collection of tasks or how many tasks you can complete in a fixed amount of time. An individual task may take longer, but this is OK if the overall time to complete a large set of tasks you care about is reduced. A classic throughput computing problem is rendering the frames of a motion picture. You don’t care if a single frame takes a bit longer, but the development team really needs to see all the frames for the next scene in a reasonable amount of time.

Regardless of whether your concern is throughput or latency, in both cases you need to precisely describe the performance of the parallel computation. We measure performance in a parallel program in terms of “speed-up”. This is the ratio of a program’s runtime in parallel to the runtime of the best available sequential implementation of that same algorithm.

\[ S = \frac{t(\text{sequential})}{t(\text{parallel})} \]

When you have a perfectly parallel program, the speed-up should scale linearly with the number of processing elements (P); i.e. S should equal P. This is called “perfectly linear speedup”. In more concrete terms, if you have perfect linear speedup, as you double the number of cores, the program runs in half the time. Programmers only rarely attain perfect linear speed-up. The problem is that even in a highly parallel problem, there are still numerous sources of overhead and this overhead limits your performance.

The most important issue to consider is the serial fraction of a problem and its impact on the speed-up. The rule governing this relation is called Amdahl’s law. Here is a quick and easy development of Amdahl’s law. Consider a problem with total runtime \( T(s) \). We can break this down into two parts, that fraction that can run in parallel (fp) and that fraction which can only run serially (fs). The serial fraction arises from operations such as I/O overhead, fixed startup costs, or work required to process the results from the parallel job. As you increase the number of processors available to work on a problem, only the parallel fraction speeds up so the time as a function of the number of processors (P) is:

\[ T(P) = (fs + fp/P)T(s) = (fs + (1/fs)/P)T(s) \]

If we plug this into the definition of speed-up we get:

\[ S = \frac{T(s)}{(fs+(1-f(s))/P)T(s)} = 1/(f(s) + (1-f(s))/P) \]
which in the limit of large \( P \) gives us:

\[
S = \frac{1}{f(s)}
\]

These equations clearly showing the impact of an algorithm's serial fraction are vital when thinking about parallel algorithms. They say, for example, that if you have parallelized only 80% of your algorithm, the best speedup you will EVER get is 5, regardless of how many processors you have. And with two processors and 80% of your problem parallelized, Amdahl's law says the best speedup you can get is only 1.4. Hence, if you care about performance, you need to pay a great deal of attention to minimizing the fraction of work in your algorithm that can only run serially.

What makes parallel programming so challenging is that in addition to performance problems arising from the parallel algorithm, the issues familiar to serial programs are still with us. For example, you must be careful to manage your memory access patterns so processing elements are close to the data they will use. This is even more important as you move to multiprocessor systems, especially as they grow in size and the costs of accessing different regions of memory vary from one processor to the next (i.e. non uniform memory architecture or NUMA systems).

**Errors unique to parallel programming**

High performance is certainly the name of the game in parallel programming. But quickly getting the wrong answers is also undesirable. Correctness in a multi-threaded program can be very challenging. The fact that the threads all share a single address space greatly simplifies parallel programming. But this feature also raises unique challenges since threads may share data in ways the programmer did not anticipate. When a program's output changes as the detailed scheduling of threads changes, we call this "a race condition". These are among the most dangerous of all programming errors since it can be almost impossible to prove that you don't have any race conditions in your program. It may run perfectly the first 1000 times, for example, but on the 1001st time, the thread schedules line up a certain way and a memory conflict causes the program to fail.

To eliminate race conditions in multi-threaded programming, the programmer must assure that every conceivable way the instructions can be interleaved yields the correct results. This means that the programmer must identify objects that multiple threads need to read or write and protect them with synchronization constructs so every order of access provides the correct results.
A deeper view into multi-threading

The rest of this section is a bit more technical. You can skip it and still sound like an expert. But if you really want to impress people with your expertise, understanding the next few paragraphs will be extremely valuable. Consider a simple program where two independent results (A and B) are computed and then combined with a third value to compute a final answer. A multi-threaded program to carry this out on two threads is represented in the figure

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = BigJob()</td>
<td>B = BigJob()</td>
</tr>
<tr>
<td>Res += A</td>
<td>Res += B</td>
</tr>
</tbody>
</table>

If the calculation of one value (e.g. A) takes much longer than the other value (e.g. B) this program would probably generate the correct result. If both threads try to sum their values into Res at the same time, however, the results are unpredictable. For example, if A = 1, B = 2 and Res is 3 (prior to the calculation of A and B), the result could be

<table>
<thead>
<tr>
<th>Final value of Res</th>
<th>Sequence of ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>if you get lucky and the A and B calculations take widely different times</td>
</tr>
<tr>
<td>5</td>
<td>if thread 1 reads Res, but while it’s doing the sum, thread 2 reads Res, does its sum and writes the result after thread 1 completes</td>
</tr>
<tr>
<td>4</td>
<td>if thread 2 reads Res, but while it’s doing the sum, thread 1 reads Res, does its sum and writes the result after thread 2 completes</td>
</tr>
</tbody>
</table>

Each of these values result from a legitimate interleaving of the instructions in the multithreaded program. But only the first one represents the correct answer. The programmer, therefore, must constrain the program so any way the instructions are interleaved results in the correct answer. This is done by protecting the update of Res with a synchronization construct.

Every multithreading API includes an assortment of synchronization constructs. With OpenMP, the natural synchronization construct to use in this case is the atomic construct. It assures that only one thread at a time will do the protected operation and that the operation with one thread will complete before the other thread is allowed to begin. The resulting program follows:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = BigJob()</td>
<td>B = BigJob()</td>
</tr>
</tbody>
</table>
While it is easy to see these problems in tiny code segments, in a real application where the memory conflicts may be buried in thousands of lines of code spread out between many different files, it can be extremely difficult to make sure you have found all of the potential memory conflicts. Similarly, finding all the sources of parallel inefficiency can be extremely challenging. The programmer must have a deep understanding of their program to deal with performance and correctness issues. The best way to do this is with the right tools. Intel has a suite of tools for analyzing the system level behavior of your program, the memory conflicts in a multithreaded program, and to visualize the parallel performance of your program. These tools are essential if you are serious about creating effective multithreaded programs.

Next Steps

You now know the key terms experts use when talking about performance (speedup and Amdahl’s law) and the big issue in correctness of parallel programs (race conditions). Next we will write some parallel programs.

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.