Parallel Scalability Isn't Child's Play, Part 2: Amdahl's Law vs. Gunther's Law

Mark B Friedman 29 Apr 2009 12:51 AM 5

Part 1 of this series of blog entries discussed results from simulating the performance of a massively parallel SIMD application on several alternative multi-core architectures. These results were reported by researchers at Sandia Labs and publicized in a press release. Neil Gunther, my colleague from the Computer Measurement Group (CMG), referred to the Sandia findings as evidence supporting his *universal scalability law*. This blog entry investigates Gunther's model of parallel programming scalability, which, unfortunately, is not as well known as it should be. Gunther's insight is especially useful in the current computing landscape, which is actively embracing parallel computing using multi-core workstations & servers.

Gunther's scalability formula for parallel processing is a useful antidote to any overly optimistic expectations developers might have about the gains to be had from applying parallel programming techniques. Where Amdahl's law can be used to establish a theoretical *upper limit* to the speed-up that parallel programming techniques can provide, Gunther's law can also model the retrograde performance that we frequently observe when parallel computing is used. In other words, Gunther's scalability formula encapsulates the behavior we frequently observe where adding more and more processors to a parallel processing workload can result in *degraded* performance. It is a more realistic model for people who adopt parallel programming techniques to enhance the scalability of their applications on multi-core hardware. So, without in any way trying to diminish enthusiasm for the entire enterprise, it is crucial to understand that achieving improved scalability using parallel programming techniques can be very challenging.

As I discussed in a review of Gunther's last book, Gunther's law adds another parameter to the well-known Amdahl's Law. Gunther calls this parameter *coherence*. Parallel programs have additional costs associated with maintaining the "coherence" of shared data structures, memory locations that are accessed and updated by threads executing in parallel. By incorporating these coherence-related delays, Gunther's formula is able to model the retrograde performance that all too frequently is observed empirically. The blue line marked "Conventional" in the chart Sandia Labs published (Figure 1 in the earlier blog) is a scalability curve that Gunther correctly cites is consistent with his model.

Let's drill into the mathematics here for a moment. What Gunther's calls his Universal Scalability law is an extension to the familiar multiprocessor scalability formula first suggested by Gene Amdahl. In Amdahl's law, p is the proportion of a program that can be parallelized, leaving $1 - p$ to represent the part of the program that cannot be parallelized and remains serial. Amdahl's insight was that the $1-p$ amount of time spent in the serial execution portion of the program creates an upper bound on how much its performance can be improved when parallelized.

Consider a sequential program that we want to speed up using parallel programming techniques. An old-fashioned way to think about this is to identify some portion of the program, p, that can be executed in parallel, and then implement a **Fork()** to spawn parallel tasks, followed by a **Join()** to unify the processing and carry on sequentially afterwards. Conceptually, something like this:

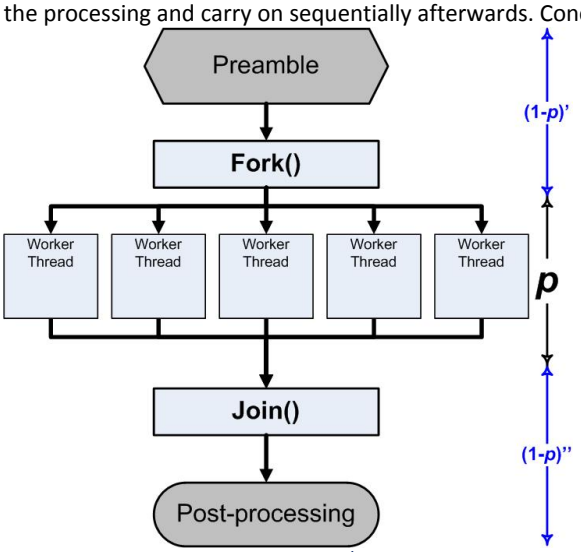

Figure 3. Parallel processing using a Fork/Join.

Amdahl's law simply observes that your ability to speed up this program using parallelism is a function of the proportion of the time, p, spent in the parallel portion of the program, compared to s, the time spent in the serial parts of the program. (Note that $p + s = 1$, in this formulation.) Amdahl's observation was meant as a direct challenge to hardware architects who were advocating building parallel computing hardware. It was also easy for those advocates of parallel computing approaches to dismiss

Amdahl's remarks since Dr. Amdahl was clearly invested in trying to build faster processors, no matter the cost.

Advocates of parallel computing, of course, are not blind to the hazards of the parallel processing approach. Scalability of the underlying hardware is one challenge. An even bigger challenge is writing multi-threaded programs. For starters, it is often far more difficult to conceptualize a parallel solution than a serial one. (We can speculate that this may simply be a function of the way our minds tend to work.) Parallel programs are also notoriously more difficult to debug. When you are debugging a multi-threaded program running in parallel on parallel hardware, events don't always occur in the exact same sequence. This is known as *non-determinism*, and it often leads to huge problems for developers because, for instance, it may be very difficult to reproduce the exact timing sequence that exposes an error in your logic.

Furthermore, once you manage to get your programs to run correctly in a parallel processing mode the performance wins of doing so are not a given. In the course of celebrating the performance wins they do get, developers can sometimes diminish an appreciation for how difficult it was to achieve those gains.

Notwithstanding the difficulties that need to be overcome, compelling reasons to look at parallel computation remain, including trying to solve problems that simply just won't fit inside the largest computers we can build. Today, there is renewed interest in parallel programming because it is difficult for hardware designers to make processors run at higher and higher clock speeds using current semiconductor fabrication technology without them consuming excessive amounts of power and generating excessive amounts of heat in the process that must be dissipated. Power and cooling considerations are driving parallel computing today for portables, desktops, and servers.

Comparing Gunther's formula to Amdahl's law.

Meanwhile, Amdahl's original insight remains relevant today. From Amdahl's law, we understand that, no matter what degree of parallelism is achieved, the execution time of a program's serial portion is a practical upper bound on the performance of its parallel counterpart. As an example, Figure 1 plots the scalability curve using Amdahl's law where p = 0.9, when just 10% of the program remains serial. Notice that Amdahl's law predicts the performance of a parallel program will level off as more and more processors are added. As you can, see Amdahl's law shows diminishing returns from increasing the level of parallelism. You can see how the parallel approach becomes less and less cost-effective as more and more processors are added.

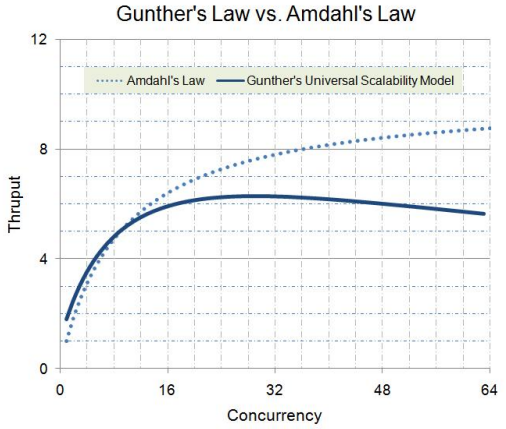

Figure 4. A comparison of Amdahl's Law to Gunther's Universal Scalability Model

Given that Amdahl was mainly acting as an advocate for building faster serial CPUs, something that he wanted to do anyway, his is by no means the last word on the subject. Researchers in numerical computing like the ones in Sandia Labs were encouraged a few years later by a paper from one of their own. John Gustafson of Sandia Labs published a well-known paper in 1988 entitled "Reevaluating Amdahl's Law" that adopted a much more optimistic stance to parallel programming. The essence of Gustafson's argument is that when parallel processing resources become available, programmers will jigger their software to take advantage of them:

> One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, *the problem size scales with the number of processors*. When given a more powerful processor, the problem generally expands to make use of the increased facilities. Users have control over such things as grid resolution, number of timesteps, difference operator complexity, and other parameters that are usually adjusted to allow the program to be run in some desired amount of time. Hence, it may be most realistic to assume that *run time, not problem size*, is constant.

Gustafson's counter-argument does not refute Amdahl's law so much as suggest there might be creative ways to work around it. It encouraged parallel programming researchers to keep plugging away, pursuing creative ways to sidestep Amdahl's law. Microsoft's Herb Sutter, in his popular Dr. Dobbs Journal column back in January 2008, cited Gustafson's Law favorably to offer similar encouragement to software developers today that need to re-fashion their code to take advantage of parallel processing in the many-core, multi-core era.

Gunther's augmentation of Amdahl's law is grounded empirically, providing a more realistic assessment of scalability using parallel programming technology. Gunther's formula is similar, but adds another parameter to Amdahl's law, κ, that represents something called *coherency* delay:

$$C(p) = p/(1+s(p-1) + kp(p-1))$$

To show how the two formulas behave, in Figure 2 above, Amdahl's law is compared to Gunther's law for a program with the same 10% serial portion. I set the coherency delay factor in Gunther's formula to 0.001. When a coherency delay is also modeled, notice that parallel scalability is no longer monotonically increasing as

processors are added. When we allow for some amount of coherency delay, there comes a point when overall performance levels off and ultimately begins to *decrease*. Gunther's formula not only models the performance of a parallel program that encounters diminishing returns from increased levels of parallelism, it also highlights the performance degradation that can occur when the communication and coordination-related delays introduced by multiple threads needing to synchronize access to shared data structures becomes excessive. Gunther's formula lumps all the delays associated with communication and coordination among threads that require access to shared data structures into one factor *k* that he calls *coherence*. Unfortunately, Gunther himself provides little help in telling us how to estimate *k,* the crucial coherency delay factor, beforehand, or measure it after the fact. Presumably, *k* includes delays associated with accessing critical sections of code that are protected by shared locks, as well as instruction execution delays in the hardware associated with maintaining the "coherence" of shared data kept in processor caches that are accessed and updated by concurrently running threads. There are also additional "overheads" associated with spinning up multiple worker threads, queuing up work items for them to process, controlling their execution, and coordinating their ultimate completion that are new to the parallel processing environment that are all absent from the serial version of the same program.

As a practical developer trying to understand the behavior of my parallel application, personally, I would find Gunther's formula much more useful if it helped me identify the sources of coherency delays my parallel programs encounter that are impacting its scalability. It would also be useful if Gunther's insight could help me guided me as I work to try to eliminate or reduce these obstacles to scalability. That is the main subject of the next blog entry in this series.

Comments

Garry Trinder 29 Apr 2009 1:31 AM #

PingBack from http://blogs.msdn.com/ddperf/archive/2009/03/16/parallel-scalability-isn-t-child-s-play.aspx

Sunny Egbo 29 Apr 2009 2:25 PM #

Parallelism exists everywhere: instruction level, memory level, loop level, task level etc. Hence, parallelism has been with us all along for quite some time now (hardware engineers have been quietly busy for the last several decades solving them for us). The major difference now is that software engineers are being asked to step up to the plate due to the fact that the brick walls created by memory speed and power have now forced CPU architects to go multi-core in a major way (2x Frequency bumps every eighteen months are no longer practical).

In practice, the theoretical treatment of computing by Amdahl, Gunther or Gustafson can obfuscate the opportunities inherent in parallel computing. These treatments often part ways from reality by glossing over a number of important points in the use of parallelism for general purpose computing, including:

1. Many user tasks are non-monolithic; they can be solved in a distributed fashion; background tasks (e.g. virus scans) in a way that improves user experiences.

2. Some algorithms have inefficient sequential solutions but surprisingly efficient parallel solutions. [The fact that many problems that have very poor sequential algorithms are easier to parallelize should be comforting to fans of algorithms.] For example, many applications require matrix multiplication, which turns out to be easily parallelizable. Although the best sequential algorithm for matrix multiplication has a time complexity of O(n2.376), a straight-forward parallel solution has an asymptotic time complexity of O(log n) using n2.376 processor. In other words, we can readily find a parallel solution for matrix multiplication that improves its runtime as more and more processor cores become available.

3. Some poor sequential algorithms can be (easily) parallelized to execute in less time than their sequential solutions. For example, bubble-sort takes (n2+n)/2 time units when programmed to run sequentially; but it takes (n2 +6n)/4 time units when programmed to run in parallel in two processors. Thus, the parallel solution to bubble sort runs better in a multi-core environment when the size of the list is greater than four elements. Parallel bubble-sort takes even less time when many more processors are available. Hence, at some point, the benefit of the simpler parallelization of some solutions can outweigh the benefit of an equivalent complex but efficient sequential algorithm

In summary, it is where the rubber meets the road, in practice, that matters.

Garry Trinder 9 Jun 2009 12:57 AM #

In the last blog entry in this series , I introduced the model for parallel program scalability proposed

Garry Trinder 9 Jun 2009 4:32 PM #

In the last blog entry in this series , I introduced the model for parallel program scalability proposed

niks 19 Dec 2014 12:50 PM #

Good article, but I don't understand how a theoretical formula could shed light on overhead that's tied to time and place by hardware, os, language, etc. Also not sure if a theoretical upper bound is often taken literally.