White Paper

Amdahl's Law, Gustafson's Trend, and the Performance Limits of Parallel Applications

By Matt Gillespie

Abstract

Parallelization is a core strategic-planning consideration for all software makers, and the amount of performance benefit available from parallelizing a given application (or part of an application) is a key aspect of setting performance goals for the parallelization process. Theoretical discussions of performance potential are necessarily the starting point for understanding the critical issues involved, before moving to practical issues associated with a given project. Amdahl's Law and its modification by Gustafson's trend give us the basic means to understand what's possible for a given application, and tools and best practices give us the means to decide how to use that information in practice.

Overview

Whatever else the future holds, processors will deliver increasing performance for the foreseeable future through parallelization, manifested by higher numbers of execution cores per processor package. There will no doubt be other dramatic enhancements, but multi-core processing is unique in the sense that both the trend itself and the means to take advantage of it (as well as the penalties for failing to do so) are highly predictable.

Software makers can be relatively assured that their efforts in threading their applications will afford them performance benefits in both the near and longer terms. Moreover, it is clear that robust threading practices will continue to become even more important as time goes on and the number of cores in mainstream processors continues to grow.

The effort to quantify the potential for performance increases by means of parallelization draws from a tradition of study that traces its roots to the work of Gene Amdahl in the 1960s. Some 20 years later, John Gustafson reconsidered Amdahl's findings, modifying those original conclusions. Taken together, these characterizations of the potential performance impacts of parallelizing software form a canonical statement of theory, with which everyone associated with software development should be familiar.

This paper provides an overview of Amdahl's Law and Gustafson's Trend, placing them in the context of current development considerations. This discussion is intended as a strategic resource for software makers and others concerned with the future of software performance in the multi-core age.

Primary Performance Considerations for Future Generations of Software

In order to illuminate software optimization as the means of taking advantage of hardware performance, it is useful to characterize advances in that hardware performance in three broad categories:

- **Increasing clock speed.** Until recently, increases in the performance of mainstream computer hardware were dominated by rising processor clock speed. Taking advantage of these incremental advances typically required no effort on the part of the software community, and such increases are no longer a significant contributor to performance.
- **Increased parallelism.** Increasing numbers of execution cores are a key means by which successive generations of processors will increase hardware performance. Developers must ensure that there software is sufficiently parallel in order to take advantage of this trend, primarily by means of multi-threading.
- **Other architectural advances.** Some architectural performance boosts are incremental (such as larger cache and faster front-side bus), while others are more quantum in nature (such as new instruction sets, pipeline improvements, and heterogeneous cores). While the former group typically requires no changes to software, the latter group sometimes does require specific optimizations.

The above list suggests that, of the factors that will increase native hardware performance going forward, developers should be concerned primarily with increased parallelism and support for

specific architectural improvements such as new instruction sets.[1] As summarized in Table 1, parallelism is the single factor that software makers can most easily plan for in the context of supporting future hardware architectures, namely by incorporating robust multi-threading into their software.

Table 1. Software performance-scaling factors for future hardware advances

Hardware performance enhancement	Action required by software developers to take advantage of enhancement
Clock speed (no longer a primary means of performance enhancement)	None
Parallelism	Increased multi-threading
Increased cache size, front-side bus speed, etc.	None
New instruction sets, pipeline improvements, heterogeneous cores, etc.	Various

Multi-threading can add significantly to the complexity, time requirements, and cost of the software development process, and so a balance must be struck between the needs of planning ahead and meeting near-term budget and time-to-market goals. While it may be superficially attractive simply to ensure that the current generation of software takes good advantage of the hardware it is likely to be run on during its life cycle, that short-term strategy can be a long-term liability.

As discussed in the paper Scaling Software Architectures for the Future of Multi-Core Computing, robust threading will become more important as time goes on, as the performance penalties for inadequate threading become more severe. Failure to put good threading practices to work now may cause software makers to perform redundant work later, driving up development costs and making their products less competitive in the marketplace.

As software companies develop a long-term strategy around parallelizing their applications, it is valuable to begin with the theoretical underpinnings of how parallelizable an application is. Next, by considering some key tools and processes that can help them to implement multi-threading in their applications, decision makers can set expectations around near-term and long-term goals.

[1] This summary conclusion is somewhat over-simplified for purposes of illustration. It does not attempt to account for factors such as entirely new hardware architectures, the need to support new virtualization technologies, or considerations of cross-platform compatibility.

Amdahl: First Steps Toward Quantifying Parallelizability

In 1967, Gene Amdahl's work at IBM began to quantify the inherent difficulties of scaling up computational performance through parallelization. He began by positing that certain "housekeeping" activities are necessary within the execution environment, which tend to be sequential (serial) in nature and therefore not able to benefit from parallel processing [1]. Because those activities represent a fairly fixed proportion of the overall computational load, Amdahl goes on; parallelizing the load in general is in vain without driving up the accompanying sequential processing rates. Moreover, he suggests, irregularities in real-world problem sets would degrade parallel performance even further.

From the outset, Amdahl shows that increasing the parallelism of the computing environment by some number *N* (e.g., providing *N* times the number of processors or cores) can never increase performance by a factor of *N.* The two main factors contribute to this limitation are the presence of the inherently serial portion of the computational load (the performance of which cannot be improved by parallelization) and the overhead associated with parallelization. That overhead consists of such factors as creating and destroying threads, locking data to prevent multiple threads from manipulating it simultaneously, and synchronizing the computations performed among various threads to obtain a coordinated result.

Successive work led to a set of mathematical relationships that became known as Amdahl's Law, which quantifies the theoretical speedup that can be obtained by parallelizing a computational load among a set number of processors. One way of expressing that relationship is given in Equation 1.

Equation 1. A general representation of Amdahl's Law

$$Speedup\ (N) = \frac{1}{S + \frac{1 - S}{N}} - O_N$$

Where:
N = Number of processor cores
S = Serial percentage of workload
(expressed as a decimal in the range 0-1)
O_N = Parallelization overhead for N threads

A simplified case of Equation 1 helps to illuminate the relationship being shown. Consider the equation with ***S*** (the serial, un-parallelizable portion of the workload) equal to zero, meaning that the workload is fully parallelizable; in this case, the speedup is equal to $N + O_N$. Further simplifying that expression by setting O_N equal to zero (removing the parallelization overhead) reduces the equation to ***Speedup (N) = N***. Therefore, for example, if one neglects both the serial component of the workload and the parallelization overhead (the ideal case), the speedup from splitting a workload from one processor onto two processors produces a speedup of 2x, splitting it onto eight cores would yield a speedup of 8x, etc. Further, viewing Equation 1 as the subtraction of O_N from a complex fraction, the complex fraction represents the speedup without being adjusted for threading overhead.

To illustrate the limitations on possible performance gains from parallelizing workloads (which was Amdahl's actual intent), consider the effect on Equation 1 when ***N*** tends toward infinity and O_N tends toward zero. That represents the case where infinitely parallel processing capacity is available, without any overhead from parallelization, and it therefore demonstrates the theoretical upper limit

to the performance increase available from parallelization. As ***N*** becomes infinitely large, the expression ***(1 - S) / N*** becomes infinitely small, so that the specialized case of Equation 1 with infinitely parallel resources and zero parallelization overhead is reduced to the expression shown in Equation 2.

Equation 2. A specialized case of Amdahl's Law with infinitely parallel execution resources and zero parallelization overhead

$$\text{Speedup (upper limit)} = \frac{1}{S}$$

Where:
S = Serial percentage of workload (expressed as a decimal in the range 0-1)

In this rarified case, note that if the inherently serial portion of the computing load is equal to 40 percent of the total load (the figure originally used as a baseline by Amdahl [1]), the maximum speedup, *regardless of the number of processor cores applied to the problem* would be ***(1 / .4)***, or 2.5x. One conclusion made by Amdahl is that "... the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude." [1] Obviously, that restriction would bode ill for the modern computing industry as a whole, and it runs counter to our intuition (with the benefit of hindsight) some 40 years later.

Gustafson: Adding Due Consideration for Large-Scale Resources and Tasks

In 1988, John Gustafson, working with E. Barsis, helped to refine Amdahl's model by adjusting some of its underlying assumptions to more accurately reflect the course of his work at Sandia National Laboratories. [2] That is, whereas Amdahl's Law indicates that the speedup from parallelizing any computing problem is inherently limited by the presence of serial (non-parallelizable) portions, Gustafson's Trend posits that this is an incomplete relationship. Gustafson argues that, as processor power increases, the size of the problem set also tends to increase. To cite one obvious example: as mainstream computational resources have increased, computer games have become far more sophisticated, both in terms of user-interface characteristics and in terms of the underlying physics and other logic.

In that video game example, consider the dramatic increase in rendering between arcade games of 1970s vintage and mainstream games of today. Since image rendering is to a large extent inherently parallel in nature (independent rendering of many blocks of pixels simultaneously, for example), that dramatic increase in processing requirements represents an equally dramatic change in the ratio of parallel-to-serial tasks in the computational load. Put simply, as compute resources increased, the problem size also increased, and the inherently serial portion became much smaller as a proportion of the overall problem. Because Amdahl's Law cannot address this relationship, Gustafson modifies Amdahl's work according to the precept (based on experimental findings at Sandia) that the overall problem size should increase proportionally to the number of processor cores ***(N)***, while the size of the serial portion of the problem should remain constant as ***N*** increases. The result is shown in Equation 3.

Equation 3. A computational representation of Gustafson's Trend

$$Speedup\ (N) = \frac{S + N\ (1 - S)}{S + (1 - S)} - O_N$$

Where:
N = Number of processor cores
S = Serial percentage of workload (expressed as a decimal in the range 0-1)
O_N = Parallelization overhead for N threads

In this equation, note first that ***S*** represents the serial proportion of the *unscaled* workload; that is, unlike in Amdahl's Law (Equations 1 and 2), ***S*** remains steady in the numerator versus denominator as a *quantity* of work, rather than as a *proportion* of the overall work. That is, while the parallel portion of the workload ***(1 - S)***[2] scales with the number of processor cores in the numerator of the equation, the serial portion ***(S)*** does not. Obviously, Equation 3 can be easily simplified by adding the components of the denominator together, and by doing so as well as eliminating (for the moment) the effect of parallelization overhead, Gustafson's trend reduces to the relationship shown in Equation 4.

Equation 4. A simpler representation of Gustafson's Trend

$$Speedup\ (N) = S + N\ (1 - S)$$

Where:
N = Number of processor cores
S = Serial percentage of unscaled workload (expressed as a decimal in the range 0-1)

Taking the most extreme case first, according to this simplified version of Gustafson's trend, scaling the number of processor cores toward infinity should result in a speedup that also scales toward infinity. Of course, infinite numbers of cores are not directly relevant to real-world implementations, but this relationship is instructive as a comparison with Amdahl's Law. To see more clearly what the effect of increasing the number of cores on a specific workload might be, consider a computational load that is 10 percent serial, where the serial portion remains a fixed size and the parallel portion increases in size proportionally to the number of processor cores, as called for in Gustafson's Trend. Table 2 shows the projected result as the number of processor cores applied to the theoretical problem is increased.

Table 2. Gustafson's Trend applied to a hypothetical problem being scaled to various numbers of processors

# cores	Computation	Speedup	Efficiency (speedup / # cores)
2	.1 + 2 (1 - .1)	1.9x	95.00%
4	.1 + 4 (1 - .1)	3.7x	92.50%

[2] Since ***S*** is a decimal representation of the percentage of the overall workload, the quantity ***(1 - S)*** represents the parallel portion.

32	.1 + 32 (1 - .1)	28.9x	90.31%
1024	.1 + 1024 (1 - .1)	921.7x	90.01%

Clearly, these calculations show that the performance result continues to scale upward as more processor cores are applied to the computational load. It's also worth noting that the per-core efficiency trends downward as additional cores are added, although the data in Table 2 shows the decrease in per-core efficiency between the two-core case and the four-core case to be greater than the entire decrease between four cores and 1024 cores. On the other hand, this relationship does not take parallelization overhead into account, which obviously increases dramatically as the number of threads (and therefore the complexity of the associated thread management) increases.

Identifying and Overcoming Real-World Limitations to Parallelization

Both Amdahl and Gustafson focus on theoretical performance scaling in the ideal case, rather than on the confounding factors that inherently limit scalability (represented as O_N in Equations 1 and 3). While it is beyond the scope of this paper to examine those overheads in depth, brief consideration of them gives the context necessary to consider how the theoretical discussion above relates to the real world. To begin, it is important to recognize that the overhead from a given number of threads is not a set quantity, but rather a reflection of the efficiency with which threading has been applied to a given piece of software (though it can never be equal to zero). Once threading has been introduced into the application, the tuning process must identify bottlenecks that represent threading overhead; broadly speaking, most elements of threading overhead fall into the following categories:[3]

- **Thread creation/destruction.** Common mitigation techniques include thread pooling (creating reusable threads) to decrease the need to create and retire worker threads, as well as threading outer loops to make a smaller number of larger loads for individual threads.
- **Synchronization and lock management.** Worker threads lock critical sections of code to protect data they are using from being overwritten by other threads in a way that could produce unexpected results. Because locking forces threads to wait for each other, increasing sequential processing requirements, locks should be used judiciously.
- **Load imbalance.** Consider the extreme case where an application spawns two threads and one performs heavy computation, while the other simply writes the result of that computation to the screen. The latter thread would be idle much of the time, decreasing the application's overall efficiency. Such issues, which are often very subtle, are an important factor in threaded application tuning.

[3] This list assumes that all threading errors have been addressed. Note that threading errors are notoriously difficult to detect, since many occur only sporadically. Intel® Thread Checker is a software development product designed specifically to help detect potential threading errors in code.

As the number of processor cores available to the workload increases, so must the number of threads. Larger numbers of threads can increase the complexity and impacts of each of the above types of overhead, driving down the efficiency of the overall code on a per-core basis. This effect is central to the need on the part of software developers to build good threading practices into their products in preparation for the projected increases in the number of cores per processor.

Conclusion

Amdahl's and Gustafson's theoretical constructs about the performance limits of parallelization are an important foundation to our understanding of how future software will manifest the power of future hardware. Placed in the context of real-world considerations about the overheads associated with software multi-threading, they illuminate the possibilities that multi-core hardware affords individual applications that have been properly parallelized.

Best practices and tools from Intel are a key means of enabling developers to build solid and scalable threading into their applications. The Intel® Software Network Multi-Core Developer Community is a key resource for those resources. Intel® Software Development Products are designed with threading in mind, including libraries of pre-threaded functions that lighten programming burdens, as well as tools that help developers identify threading errors and performance issues, speeding up the development process and increasing the quality of threaded applications.

References/Additional Resources

The following materials, some of which are referred to in the text of this paper, provide a point of departure for further research on this topic:

[1] G.M. Amdahl, Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

[2] John Gustafson, Reevaluating Amdahl's Law, Communications of the ACM 31(5), 1988; reposted at http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html.

[3] Herb Sutter, The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb's Journal, 30(3), March 2005.

[4] Michael Wrinn, Is the free lunch really over? Scalability in Many-core Systems, Intel Software Network.

About the Author

Matt Gillespie is an independent technical author and editor working out of the Chicago area and specializing in emerging hardware and software technologies. Before going into business for himself, Matt developed training for software developers at Intel Corporation and worked in Internet Technical Services at California Federal Bank. He spent his early years as a writer and editor in the fields of financial publishing and neuroscience.