Dispelling the Myths of Parallel Computing

Patrick H. Madden
SUNY Binghamton Computer Science Department

Editors' notes:
Professor Patrick Madden presents his (agnostic) take on parallel computing. We believe that having an agnostic view allows us to focus on the deficiencies of the systems we are building, so we can improve on them.
—Rasit Onur Topaloglu, IBM, and Beven Baas, University of California, Davis

For decades, the performance of serial computing systems has grown by leaps and bounds. Driving this progress has been Moore's Law; the number of transistors available has rapidly increased while costs have plunged. Architecture innovations and increased core clock frequencies have led to explosive growth, but limits appear to have been reached: the 2001 International Technology Roadmap for Semiconductors (ITRS) [21] projected clock frequencies scaling from 1.684 GHz in 2001 to 6.739 GHz by 2007, but because of a variety of challenges (primarily power constraints), modern microprocessors fall far short of this mark.

As an alternative to serial performance gains, the industry has turned towards parallel computation—increasing the number of processor cores, while keeping core clock frequencies relatively stable. Competition between processor vendors now commonly features core counts, rather than clock rates. The 2009 ITRS [22] projects the number of cores doubling every other technology generation.

Parallelism is by no means a new idea, and for many years there has been debate on how much benefit can be derived [1], [2], [12], [20]. With little hope for further serial performance gains, software developers are now forced to embrace parallelism, and will naturally look to the scientific literature to find best practices.

There are a number of obvious parallel computing successes, but there are also a great many pitfalls, and some surprisingly large errors in the literature. A primary objective of this paper is to highlight myths and misconceptions, so that past mistakes can be avoided.

Digital Object Identifier 10.1109/MDT.2012.2230391
Date of publication: 29 November 2012; date of current version: 11 April 2013.

Background

In this section, we will define a few terms, and briefly recap computational complexity theory. While most computing professionals know this material thoroughly, we will summarize the topic, as it is critical to dispelling common myths.

Parallel scalability and serial constraints

In order for a parallel system to complete a task more quickly than a serial system, there must be work that can be divided up to keep multiple processors busy. It's not difficult to find this: if one were to add vector A to vector B, each of the $A_i + B_i$ terms could be computed simultaneously.

We will refer to an algorithm as *scalable* if it can be accelerated linearly to large numbers of processors. Vector addition, as well as many other tasks, are scalable.

At the same time, however, there are many tasks with serial constraints. Some value X might be necessary before beginning a computation for Y, which is in turn needed by Z. Serial constraints between operations limit scalability.

Copublished by the IEEE CEDA, IEEE CASS, IEEE SSCS, and TTTC

Computational complexity

Complex software applications, whatever they might be, are algorithms. In practice, an application has a great many subcomponents, many of which perform well-known tasks (searching, sorting, and so on). Algorithmic comparisons are normally done using the computational complexity framework developed by Hartmanis and Stearns [15].

As a simple example, consider the problem of finding $\sum_{i=1}^{n} i$, the sum of the integers from 1 to n. This can be accomplished with a simple loop construct; the computational complexity of such an approach is $O(n)$, with run time growing linearly with n. A second approach to consider is the closed form, $(n \times (n+1))/2$. The closed form is $O(1)$, constant time—it should be obvious that no matter how slowly one might perform the multiplication and division steps, there is a value of n at which the closed form is guaranteed to overtake the loop (and in practice, that value is small).

This basic principle applies to more complex algorithms. Sorting, for example, is well studied, and there are a number of well known sorting algorithms [6]. In Table 1, we show the run times for sorting between 10,000 and 160,000 randomly generated integers, using a variety of $O(n^2)$ and $O(n \log n)$ algorithms. An ordinary 2.4 GHz ×86-based laptop with the Gnu C compiler was used for this set of experiments.

The results illustrate the connection between computational complexity and run time; doubling the size of the input produces roughly a factor of four increase in run time for the $O(n^2)$ algorithms. Similarly, the run times of the $O(n \log n)$ algorithms increase slightly more than linearly with problem size.

Source code for the sorting algorithms mentioned above is easily available; interested readers are encouraged to try running the experiments for themselves. For the next section, it will be useful to keep in mind the run-time differences between $O(n^2)$ algorithms and $O(n \log n)$ algorithms.

Table 1 Run times (in seconds) for $O(n^2)$ and $O(n \log n)$ sorting algorithms, along with the run time ratios for each $O(n^2)$ sorting algorithm compared to quick sort.

	Problem Size				
	10k	20k	40k	80k	160k
Bubble Sort	0.474s	1.872s	7.460s	30.210s	121.642s
vs. QS	237x	468x	746x	1438x	2644x
Selection Sort	0.377s	1.484s	5.976s	24.013s	95.748s
vs. QS	188x	371x	597x	1143x	2081x
Insertion Sort	0.258s	1.013s	3.885s	15.225s	60.972s
vs. QS	129x	253x	388x	725x	1325x
Rank Sort	0.948s	3.816s	15.241s	61.233s	245.293s
vs. QS	474x	954x	1524x	2915x	5332x
Merge Sort	.002s	.004s	.011s	.022s	.048s
Heap Sort	.003s	.006s	.015s	.033s	.069s
Quick Sort	.002s	.004s	.010s	.021s	.046s

The algorithmic landscape

An algorithm for a given problem is considered *efficient* if there are no other known algorithms for the problem with lower computational complexity. When developing a software application, a number of different algorithms for any given task may be available. Figure 1 shows a Venn diagram, grouping algorithms by efficiency and scalability.

Some algorithms, such as bubble sort, are neither efficient, nor scalable. Others, such as the naive approach to computing the sum from 1 to n, are scalable, but inefficient. There are a number of sorting algorithms (quick sort, heap sort, merge sort) that are efficient, but have limited scalability at best. Finally, there are algorithms such as vector addition that are both computationally efficient, and also scalable to large numbers of processors.

Note that there is always a serial implementation of the most efficient algorithm for any problem (a single processor can emulate multiple processors,

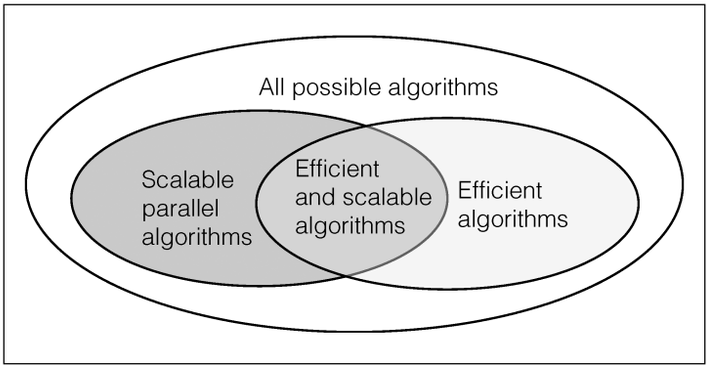

Figure 1. A Venn diagram relating efficient algorithms and scalable parallel algorithms. Each of these sets is nonempty, with well known examples. Bubble sort is neither scalable nor efficient. Many sorting algorithms (quick sort, heap sort, merge sort) are efficient, but not scalable. The naive summation of 1 to *n* is scalable, but not efficient. A simple parallel vector addition approach is both efficient and scalable.

with no more than a constant factor overhead). Further, from the run times illustrated in Table 1, it should be clear that when implementing an application, only efficient algorithms should be considered for large problems.

Myths of parallel computing

In this section, we will focus on the limits of parallel computing, and highlight a number of published works where popular myths have caused authors to overlook significant errors.

Myth: High scalability implies high performance

The first myth to consider is the idea that scalability—the ability of an application to utilize large numbers of processors to reduce run time linearly—is equivalent to high performance. Scalability is sometimes measured as processor utilization, or by the number of Floating Point Operation per Second (FLOPS) achieved.

From the brief consideration of complexity in the previous section, a pitfall should be apparent. The naive method to compute $\sum_{i=1}^{n} i$ with a simple loop is scalable; the summation can be split across an arbitrary number of processors, giving almost limit less speed-up for large values of n. The closed form, however, is faster—despite not being scalable, and utilizing only a single processor.

When considering sorting algorithms, a similar issue arises. The rank sort [19] algorithm makes a comparison between each pair of elements, to determine the relative position of each in the sorted output. This can be made massively parallel easily, with the approach being scalable to large numbers of processors. With a parallel system (and assuming no overhead for communication, synchronization, or memory access), for a problem with 10,000 integers, one might need close to 500 processors to match the run time of a single processor using Quicksort. If the problem size increases to 160,000 integers, one would need more than 5,000 processors. It should be clear that even a massively parallel rank sort would be uncompetitive with a simple serial quick sort for large data sets—and for small data sets, there would not be enough work to distribute to each processor to make doing so worthwhile.

Further, it should be noted that while parallelism might recover some of the run time loss due to algorithmic inefficiency, *it does not recover power consumption losses*. A less efficient algorithm executes substantially more instructions—resulting in a dramatic increase in total power use.

It might seem absurd to use a less computationally efficient algorithm as a means to enhance parallelism—but in fact, a surprising number of published works suggest doing exactly this.

Specific example: shortest path algorithms. Computing a shortest path between a pair of points in a graph is a common task in circuit routing (as well as a number of other applications); there are two well known algorithms for this, a $O(n^2)$ dynamic programming based approach by Bellman and Ford, and a $O(n \log n)$ approach by Dijkstra [7].

In a 2008 tutorial on General Purpose Graphics Processing Units (GPGPUs) presented at the Design Automation Conference [10], one example utilized a massively parallel version of the Bellman-Ford algorithm. For the purposes of illustrating a parallel computing model software model, this is a reasonable choice; the computational complexity impact of the approach, however, can be entirely lost on the casual reader.

From the previous sections, the results of actual runs shown in Table 2 should not be at all surprising [9]. One might expect that massive parallelism would allow for much faster path computations; in reality, the scalable parallel solution is far slower.

It might be easy to dismiss this as an anomaly-but in fact, this type of error is surprisingly common. For example, one research group implemented a custom circuit using an FPGA to solve the shortest path problem [23]—but failed to implement an efficient priority queue, resulting in a massively parallel $O(n^2)$ solution. The authors did not make any comparison with an efficient serial implementation of Dijkstra's algorithm, preventing them from recognizing the misstep.

Table 2 Shortest path computation times comparing a GPGPU with a parallel implementation of the $O(n^2)$ Bellman-Ford algorithm, and a serial CPU using either Bellman-Ford or the $O(n \log n)$ Dijkstra algorithm.

	CPU Bellman-Ford	GPGPU Bellman-Ford	CPU Dijkstra
Bunny (35k vertices)	2.9s	0.06s	0.08s
Buddha (544k vertices)	62.5s	3s	1.6s

More surprising is a doctoral dissertation [14], in which a custom VLSI circuit was designed and fabricated. The hardware design implemented a variation of the Bellman-Ford algorithm; comparisons with a software implementation of Dijkstra's algorithm were profoundly flawed due to an obvious error in the implementation.

We limit the discussion of shortest path to three examples, due to space constraints; a number of other similar errors are easy to find in the open literature. While Bellman-Ford is less efficient than Dijkstra's algorithm, it has the advantage of being applicable to graphs with negative edge weights; if a graph has negative weighted edges, then a parallel implementation of Bellman-Ford is a good solution.

Specific example: lithography and n-body problems. As a second algorithmic example, we will consider the problem of rectangle overlap detection as part of processing a design for lithography. Circuit designs can contain hundreds of millions of rectangular areas that may overlap in the construction of a mask—detecting these overlaps is an important step.

In a recent paper from an industry research group [16], a massively parallel approach to detecting overlaps between rectangular regions of a lithographic mask was presented. Each rectangle in a design was compared to every other, resulting in a straight-forward $O(n^2)$ approach. While the parallel solution provides a massive speedup of the brute force algorithm, there are computationally efficient $O(n \log n)$ methods based on computational geometry.

To implement a faster serial method, one first sorts the rectangles by their x and y coordinates; these can then be arranged into a tree-like structure, or embedded into a number of smaller tiles. By arranging the rectangles into an appropriate data structure, vast numbers of potential overlaps can be ruled out, resulting in a far more computationally efficient approach.

The underlying problem is similar in many ways to *n-body* problems found in physics. While one might implement a naive $O(n^2)$ approach to compute forces between all pairs of objects, there are more efficient methods [5]. Despite the massive advantage of the efficient algorithm, one can find many instances of the naive parallel implementations in the supercomputing community (e.g., [13]).

Another related problem is in collision detection, for applications such as video games with large numbers of independently moving objects. A "cure for the multicore blues" was described [11], in which a naive all-pairs solution was presented—as with the other examples, the scalable parallel solution is $O(n^2)$, while the reasonable serial approach is $O(n \log n)$.

It's not wrong, but it's not right either. Strictly speaking, the research cited in this section is not *wrong*; the parallel implementations are in fact scalable, obtaining reduced run times over serial implementations. What is easily missed, however, is that the scalable parallel algorithms are slower and consume more power than *efficient* serial algorithms—by a factor bounded only by the problem size. For the tasks mentioned (and many others), there are no known algorithms that are both scalable and efficient.

This misconception has been noted before. In a well known paper in the supercomputing field, Bailey [3] detailed "twelve ways to fool the masses when giving performance results on parallel computers." The use of inefficient algorithms to boost scalability is only one of these ways. Bailey's paper took a tongue-in-cheek approach to highlight very serious problems—but despite the best efforts of many researchers in the supercomputing field, the problems persist.

Myth—The parallel section is most important

A second myth to consider is the notion that high performance can be achieved by focusing on sections of parallel code, and that other sections of an application can be dismissed. The fundamental problem with this view has become known as Amdahl's Law [1]—a simple limit case where one assumes that a parallel portion P of an application can be accelerated linearly with k processors. The remainder of the run time, $S = 1 - P$, is assumed to be serial in nature.

With k processors, the total run time of an application is $S + (P/N)$. As the number of processors increases, run time converges to S. If only 10% of an application is serial in nature, it is impossible to obtain more than a factor of 10 improvement in run time. In practice, there is almost always a portion of

Typo in the final draft: equation should be S + (P/k)

an application that is serial in nature (for example, each of the cases covered in the prior subsection have limited scalability if one uses an efficient algorithmic approach).

To say that Amdahl's Law was not warmly received would be an understatement. The law paints a very bleak picture for parallel computing efforts. The dogmatic support for parallel computing by Amdahl's peers can be inferred from the opening statement of his talk:

> For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. Variously the proper direction has been pointed out as general purpose computers with a generalized interconnection of memories, or as specialized computers with geometrically related memory interconnections and controlled by one or more instruction streams.

As a means of circumventing the law, an unusual method of measurement has become common. By reporting *only* the run time of the parallel sections, one can obtain the *appearance* of scalability. To provide a specific real-world example, we refer to a demonstration program distributed with the IBM Cell processor Software Development Kit.

The *matrix_mul* program performs matrix multiplication—a task with a great deal of inherent parallelism. For large matrices, the program reports that the Cell processor achieves 23.85, 42.93, and 71.55 GFlops, for 1, 2, or 4 processing units, respectively—near linear speedup. If one inspects the source code, one can see that the time required to transfer data between processing units, or to synchronize results, is omitted.

The actual run time improvement (when one counts both the serial and parallel work required) is only 13% when moving from one processing element to two. Moving to four processing elements provides an additional 7% gain. While it might appear that one could achieve near linear speedup on a problem that clearly has a great deal of parallel potential—the reality is rather modest gains, with benefit diminishing quickly.

Myth—Scalability can be achieved with sufficient effort

The final myth we consider is the notion that challenges to scalability can be achieved by simply working harder—having programmers be more innovative, developing new architecture designs that minimize communication delays, and so on.

While skilled programmers and clever architecture designers are certainly valuable, we would note that the algorithms available play a critical role. If one needs to compute a shortest path, for example, even a Herculean effort will obtain only a modest improvement over a reasonable serial implementation of Dijkstra's algorithm.

The degree of scalability possible for an algorithm is a function of the algorithm itself. If the most efficient algorithm for a problem is serial in nature, or has limited parallelism, no amount of effort can make it scalable. A more formal description of these limits are Leiserson's *work* and *span* laws [18]. *Work* is analogous to the computational complexity of an approach. The *span* of an algorithm is in some sense the length of the "critical path" of logical dependencies. For many important problems, the most efficient algorithms have spans that are comparable to the work—preventing large scale parallelism. Even when the work and span measurements are of different orders, gains can rapidly diminish.

The quick sort algorithm, for example, can be accelerated; it uses a divide-and-conquer approach, allowing parts of the data to be sorted in parallel following an initial partitioning. The partitioning function, however, contributes to the *span* of the algorithm—it is nearly impossible to achieve more than a factor of ten speedup for even exceptionally large problems [18].

The algorithmic barriers to parallelism are well known to those who study the area carefully. One of the most noted researchers in algorithms, Prof. Donald E. Knuth, recently made the following comment [17].

> During the past 50 years, I've written well over a thousand programs, many of which have substantial size. I can't think of even five of those programs that would have been enhanced noticeably by parallelism or multithreading.

For more than half a century, considerable effort has been invested to move parallel computing from a niche technology, and into the mainstream. There is tremendous intuitive appeal for the idea—the limited success might be somewhat puzzling. A survey of the field from close to 20 years ago, noted the following [8]:

> A decade ago, university researchers were in love with parallel computers, and the U.S. government amorously responded. Those were the days of glory, but times have changed: the market for massively parallel computers has collapsed, and many companies have gone out of business, but the researchers are still in love with parallel computing.

In this paper, we have argued that a number of myths have fueled unreasonable expectations. First, we noted that the computational complexity of an algorithm is of primary importance—this was well established years ago, but seems to be overlooked. By mistaking scalability for performance, many authors have "found" abundant parallelism with inefficient algorithms, failing to realize that their proposed solutions are far slower than conventional serial methods.

The necessity of efficient algorithms unravels the second myth. If even a portion of the work to be performed requires algorithms that are serial in nature, or have limited opportunity for parallel speedup, Amdahl's Law comes in to play. As tempting as it might be to focus on sections of an application with tremendous scalability, it is the serial portions that ultimately limit gains. There is little reason to suspect that there are many efficient, scalable algorithms waiting to be discovered—many of the techniques used to achieve algorithmic efficiency introduce serial constraints.

The final myth we considered is the notion that scalability can be achieved through effort, and that by simply investing more time, energy, and funding, we can overcome the limitations of Amdahl's Law. Unfortunately, the limits imposed are fundamental to the mathematics of computation; no amount of effort will turn an unscalable problem into a scalable one.

To demonstrate that these are not hypothetical errors, or cases that occur at the margins of the computing field, we have noted papers from both academia and industry, drawn from top IEEE and ACM conferences and journals, a doctoral dissertation from a major university, a university level parallel programming text, and a software development kit for a parallel processor. We have restricted our algorithmic examples to cases where a massively parallel solution is $O(n^2)$, while the efficient approach (with limited parallelism) is $O(n \log n)$—most of the errors noted are obvious from the text of the papers, and do not require access to either source code or benchmark data. The algorithms considered are not difficult to implement or obscure; Dijkstra's algorithm, for example, is standard fare for undergraduate computer science programs. For the Cell processor example, the source code was available, allowing for direct inspection. While we have been rather blunt in our critique, there does not appear to be deliberate deception in any of these instances. Because of space constraints, we have restricted our discussion to only the most obvious cases where the errors are easy to identify.

While no research field is free of problems, the magnitude of the errors we have addressed here may be surprising. As evidenced by the struggles of the supercomputing community over many years [3], [4], eliminating these types of errors is exceptionally difficult.

The philosopher George Santayana noted "those who cannot remember the past are condemned to repeat it." We would suggest that one lesson that can be learned from the history of parallel computing is that there are pitfalls that can ensnare nearly anyone. Many times, solutions that appear to offer tremendous gains are in fact little more than illusions.

Despite these challenges, there is no choice but to forge ahead with parallelism—there is little hope for further serial performance gains. For a handful of niche areas, there are already well established parallel solutions; outside of this, the future looks much more difficult. To make progress, the research community will need to focus on solutions that truly advance the state of the art. ■

■ References

[1] G. M. Amdahl, "Validity of the single-processor approach to achieveing large scale computing capabilities," in *Proc. AFIPS Conf.*, 1967, pp. 483–485.

[2] G. M. Amdahl, Arvind, J. Gustafson, R. Goering, P. H. Madden, K. Olukotun, and G. Smith, "Can we still

keep the faith? A debate on the future of multi-core systems,'' in *ACM SIGDA Member Meeting at ICCAD,* Nov. 2007.

[3] D. H. Bailey, ''Twelve ways to fool the masses when giving performance results on parallel computers,'' *Supercomput. Rev.*, pp. 54–55, Aug. 1991.

[4] D. H. Bailey, ''Misleading performance claims in parallel computations,'' in *Proc. Design Automation Conf.*, 2009.

[5] J. Barnes and P. Hut, ''A hierarchical $O(N \log N)$ force-calculation algorithm,'' *Nature*, 1986.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithm*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[7] E. Dijkstra, ''A note on two problems in connexion with graphs,'' *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[8] B. Furht, ''Parallel computing: Glory and collapse,'' *IEEE Computer*, vol. 27, no. 11, pp. 74–75, 1994.

[9] M. Garland, *Personal Communication*, 2008.

[10] M. Garland, ''Sparse matrix computations on manycore GPU's,'' in *Proc. Design Automation Conf.*, 2008, pp. 2–6.

[11] H. Goldstein, ''Cure for the multicore blues,'' *IEEE Spectrum*, vol. 44, pp. 40–43, 2007.

[12] J. L. Gustafson, ''Reevaluating Amdahl's law,'' *Commun. ACM*, vol. 31, no. 3, pp. 532–533, 1988.

[13] T. Hamada and T. Iitaka, *The Chamomile Scheme: An Optimized Algorithm for N-Body Simulations on Programmable Graphics Processing Units,* Mar. 2007. [Online]. Available: http://arxiv.org/abs/astro-ph/0703100v1

[14] P. M. Hansen, ''Coprocessor architectures for VLSI,'' Ph.D. thesis, Computer Science Department, University of Califoria at Berkeley, Berkeley, CA, USA, 1988.

[15] J. Hartmanis and R. E. Stearns, ''On the computational complexity of algorithms,'' *Trans. AMS*, vol. 117, pp. 285–306, 1965.

[16] D. A. Jamsek, ''Designing and optimizing compute kernels on NVIDIA GPUs,'' in *Proc. Asia South Pacific Design Automation Conf.*, 2009, pp. 224–229.

[17] D. E. Knuth and A. Binstock, *Interview With Donald Knuth,* InformIT.com. [Online]. Available: http://www.informit.com/articles/article.aspx?p=1193856

[18] C. E. Leiserson, ''The cilk++ concurrency platform,'' in *Proc. Design Automation Conf.*, 2009.

[19] B. P. Lester, *The Art of Parallel Programming*, 2nd ed. Singapore: 1st World Publishing, 2006.

[20] C. M. Pancake, ''Is parallelism for you?'' *IEEE Comput. Sci. Eng.*, vol. 3, no. 2, pp. 18–37, Jun. 1996.

[21] Semiconductor Industry Association, International Technology Roadmap for Semiconductors, 2001.

[22] Semiconductor Industry Association, International Technology Roadmap for Semiconductors, 2009.

[23] M. Tommiska and J. Skytta, ''Dijkstra's shortest path routing algorithm in reconfigurable hardware,'' in *Proc. 11th Conf. Field-Programmable Logic and Applicat.*, 2001, pp. 653–657.

Patrick H. Madden is an associate professor of computer science at the State University of New York at Binghamton. His primary research areas are integrated circuit physical design, protein identification, and optimization. He has a PhD in computer science from UCLA in 1998. He is the past chair of ACM/SIGDA, has been part of the steering committees and program committees for a number of EDA conferences, was an associate editor for IEEE Transactions on Computer Aided Design, and is currently an associate editor for *ACM Transactions on the Design Automation of Electronic Systems*.

■ Direct questions and comments about this article to Patrick H. Madden, SUNY Binghamton Computer Science Department, Box 6000, Birmingham, NY 13902; phone 607-777-2943; pmadden@acm.org.

Second typo: should be Binghamton, NY