Parallel Computing: The Elephant in the Room

Patrick H. Madden
SUNY Binghamton Computer Science Department
pmadden@acm.org

August 19, 2010

Abstract

Over the past few years, there has been a shift towards multi-core processors, driven partially by physical limitations. Mistaken assumptions of how effective and useful parallel systems can be have also provided motivation for this change.

In this paper, we seek to directly identify the barriers to parallel computation. The barriers are not, as conventional wisdom might hold, a lack of programmer ingenuity or research funding. Rather, the limitations are fundamental to the problems addressed; new computer architectures and compilers offer limited benefit.

1 Introduction

In this paper, we take a critical look at parallel computing, and make three specific points. First, we wish to make clear that the laws of mathematics still apply – the availability of parallel machines does not alter the mathematical foundation of the field. Second, because of this foundation, there are limits to how useful parallel machines can be – this has serious implications for the computer industry. Third, we highlight how this foundation is overlooked, leading to profound errors in the literature, with the research field moving in the wrong direction, and focus being placed on the wrong problems.

The points we make are somewhat at odds with popular opinion. Many are optimistic about the future of parallel computing, and certainly the shift towards parallel computing has not been made lightly. This change has been motivated by a number of compelling reasons. One motivation is the apparent end of significant serial performance gains. Limited by power constraints, clock rate increases have slowed. Architectural enhancements also have diminishing returns; gains from branch prediction, caching schemes, and so on, seem nearly exhausted. Despite an abundance of silicon area, it is difficult to make a "faster" processor. A second motivation is that the design of larger and more complex monolithic circuits has become prohibitive. It is simply not practical to build (and verify) a processor as large as fabrication technology will allow.

Parallel computing seems to offer an easy solution, providing more performance, lower power consumption, and easier design. There are numerous calls for universities to focus on teaching parallel programming, and a great deal of funding support for new languages and architectures – but this is far from the first time this has occurred.

The 1968 ACM Computing Curriculum[6], for example, lists multiprocessing and multiprogramming as key topics for research. A survey in the mid 1990's[37] found that many major universities offered seven or more courses on parallel computing, with Purdue topping the list at twelve. Literally hundreds of parallel programming languages have been developed in the past half century. Thousands of research projects have been funded. Scores of parallel machines have been built. Despite all this, parallel computing can be found in relatively few areas: supercomputing, high volume servers, and in graphics applications. Elsewhere, software is predominantly serial.

Even in supercomputing, parallel computing has had difficulty. In a survey paper from 1994, Furht[12] noted the following:

> A decade ago, university researchers were in love with parallel computers, and the US 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.

One might ask, given the effort expended, *why has parallel computing not become commonplace?* There are a few tempting answers to this, but they are misleading. Some suggest that programmers lack the vision and ingenuity to implement parallel algorithms. We disagree with this notion, and refute it in the following section. A second common assertion is that there has been limited access to parallel machines – but anyone who has studied computer history knows this to be incorrect[27, 26, 22]. University and industry research groups have been building parallel machines for decades, with numerous forays into personal computing, business applications, and supercomputing (where there has been success). A third common suggestion is that consumers are not interested in the performance leaps possible with parallel machines, and preferred to only keep pace with Moore's Law[38]. Given the continual demand for faster machines, this also is questionable.

There has been active debate on the role of parallel computing for many years[1, 17, 2]. The goal of this paper is to squarely address the constraints that have kept parallel computing out of the main stream – to acknowledge the proverbial "elephant in the room." We suspect that there are many who are aware of these issues, and share our concerns, but are reluctant to speak out. The time is ripe, however, to deal with these problems openly; performance gains through serial computation have dropped dramatically, signaling a profound change in the computer industry.

2 The Elephant in the Room

There is a well known fable from India of a group of blind men who encounter an elephant. One, grasping the tail, declares an elephant to be like rope. A second touches the animal's leg, and concludes that the elephant is like a tree. A third, touching the ear, is confident that the elephant is like a fan. While there is an element of truth in each assertion, the moral of the story is that one must see something in it's entirety to properly understand.

This principle underlies a key observation by Dr. Gene Amdahl, during a talk in 1967[1]. If one views a software application as being composed of a number of smaller algorithmic parts, it can be seen that some portions may be accelerated with additional processors, but others might not. The nature of the problems addressed in each individual portion of an application impacts how effective parallel computation can be.

Algorithm 1 Vector addition, which can be made parallel easily.

```
for $i$ from 0 to $n$ do
    $A[i] = B[i] + C[i]$
end for
```

Algorithm 2 Computation of the Fibonacci sequence using a simple dynamic programming approach. Each $F[i]$ must be computed in sequence, leaving little opportunity for parallel computation.

```
$F[0] = 1$
$F[1] = 1$
for $i$ from 2 to $n$ do
    $F[i] = F[i-1] + F[i-2]$
end for
```

To illustrate two different types of computational problems, consider the code segments shown in Algorithms 1 and 2. In the first, there is an obvious way to exploit additional processing cores – this is simple vector addition, a common task. Each addition could be performed on an independent processor; given an adequate memory subsystem, there is an enormous potential for acceleration.

The second example, however, is more problematic. This code illustrates a classic method for computing the Fibonacci sequence, and is serial in nature. Before $F[4]$ can be computed, $F[3]$ must be finished. $F[5]$ depends on $F[4]$, $F[6]$ depends on $F[5]$, and so on. There is no processor architecture or interconnection network that can provide meaningful speedup. Short of fundamentally changing the algorithmic approach, a new compiler or programming language will have negligable impact.

Amdahl's observation after examining a number of programs was that while some sections were amenable to parallel computing, others were not. Seeing a program *in it's entirety* provided an important insight into how much acceleration was possible with additional processors. This has come to be known as Amdahl's Law.

A simple example can be constructed, where one assumes that a portion p of an application can be accerated with parallel computing – in other words, the structure of the program is similar to Algorithm 1. The remainder of the application $s = 1 - p$ is serial in nature, similar to Algorithm 2. The serial portion presents a bound on the maximum speedup achievable relative to a single processor. With k processors, we have the following upper bound.

$$\text{speedup} = \frac{1}{s + p/k}$$

If only 10% of an application is serial in nature, there is at best a ten-fold speedup, even with an infinite supply of additional processors. While the portion of a software application that is serial in nature can vary quite a bit, it is generally significant. Rather than seeing a potential for great performance gains, it appeared that a performance gain of more than a factor of two or three would be uncommon.

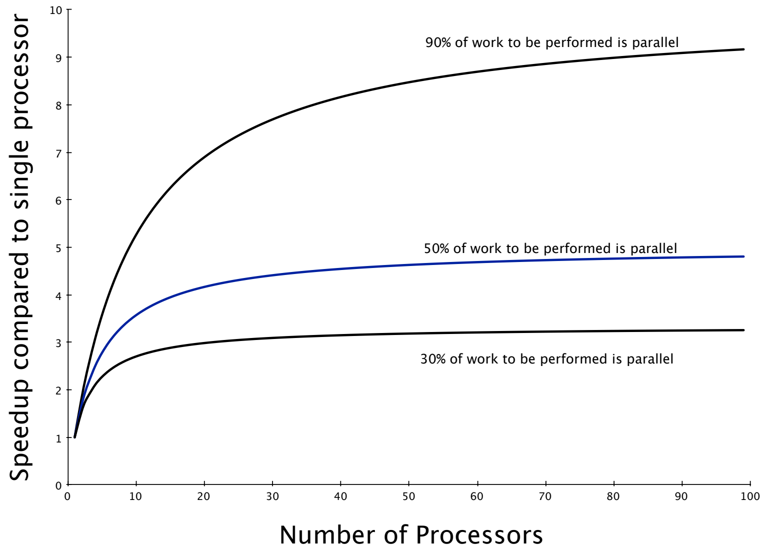


Figure 1: An illustration of Amdahl's Law. Applications typically contain code segments that are parallel, and also segments that are serial. The serial portions place an upper bound on possible performance gains.

This is illustrated in Figure 1. Amdahl's formulation is essentially a simple limit case, where the serial and parallel sections are distinct, and the parallel sections can be accelerated linearly with the number of processors (with no overhead).

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 his 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.

Despite years of effort to overturn the Law, it remains as problematic as ever (and has been significantly refined through Leiserson's Work and Span Laws[31, 10]). In the next few sections, we will illustrate how the impact of the law is grossly underestimated – to do this, a brief review of basic computer science theory is helpful.

3 Background: Computational Complexity

In this section, we will briefly recap computational complexity theory. While most computer scientists know this material thoroughly, summarizing it may be helpful to those outside the field.

Beginning with Turing's profound leap[44], methods for automated information processing have exploded. The "Universal Turing Machine" is a simple computational model with great power – Turing showed that his model could compute anything that was computable. The von Neumann architecture[21] is the most common implementation of Turing's ideas. While Turing considered parallel implementations, it was clear that a simple serial implementation was "enough." Combining machines to work in parallel could provide at most a constant factor acceleration.

Von Neumann machines (as well as any other comparable implementation that falls within Turing's mathematical universe) allow the implementation of algorithms. An algorithm is simply a series of rules and operations to be applied to a set of data; it is possible to apply multiple operations simultaneously, but this gains only a constant factor speedup.

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). In this paper, we will focus on subproblems which are well defined, but the observations apply to any software application.

Based on a framework of Turing's approach, the work of of Hartmanis and Stearns[20], established computational complexity as a metric for comparing algorithms. With complexity theory, it became possible to estimate the growth in run time of an algorithm, independent of the computer hardware used.

"Big-O" notation has become the common way to evaluate an approach; while one can assume that any competent computer scientist would be quite familiar with it, this is not necessarily the case with those outside the field. Notation varies slightly; we will use the formulation from the Cormen[10] text. Formally, one may count the number of steps an algorithm performs as a function $f(n)$, where n is the size of the problem under consideration. A function $f(n)$ belongs the set $O(g(n))$ if $0 \leq f(n) \leq c \times g(n)$ for some constant $c > 0$ and all n larger than some value n_0. There is a similar lower bound function, $\Omega(g(n))$, and if the upper and lower bound functions are identical, we have $\Theta(g(n))$.

Any given task may have a variety of different algorithmic approaches. Using "big-O" complexity analysis, it is possible to compare different approaches, and determine *with certainty* which approach will be fastest for large problems. The great value of this theory is that the best algorithm can be determined without needing to know anything about the hardware on which it is implemented. We can plot the number of operations (or run time) versus problem size; if the two computational complexity functions differ, their run time curves will diverge.

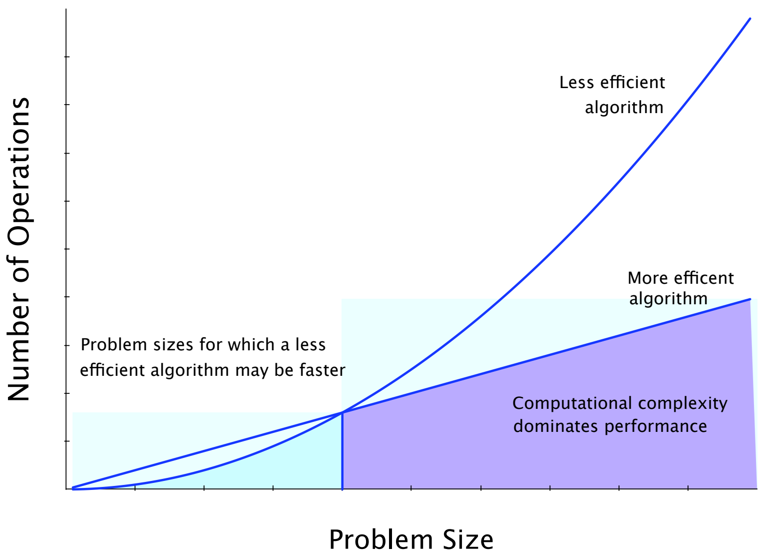


Figure 2: The theoretical implications of computational complexity; if two algorithms with different complexities are implemented, the one with the lowest growth rate will be fastest for problems larger than some size n_0, regardless of the constant factor impact of processor speeds.

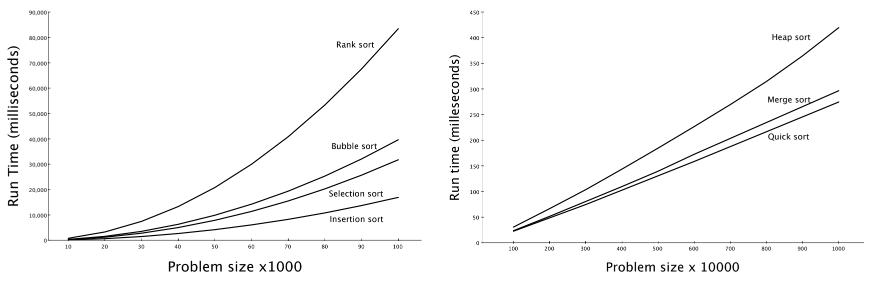
This is illustrated in Figure 2. If, for example, we have one algorithm that is $\Theta(n^2)$, and a second that is $\Theta(n \log n)$, it is possible (but not guaranteed) that there exists a value of $n_0 > 0$ such that the $\Theta(n^2)$ algorithm is faster than the $\Theta(n \log n)$ algorithm. For large values of n, however, the $\Theta(n^2)$ algorithm quickly becomes uncompetitive, and the margin by which it is slower is not a simple constant – it grows in an unbounded fashion. Further, each operation performed consumes some amount of power; the $\Theta(n \log n)$ algorithm will also be more power efficient, no matter what computing systems are used.

Complexity theory is not an outdated concept; to illustrate this we will present experimental results using a few classic sorting algorithms. Figures 2a and 2b show run times for sorting large sets of integers. The first graph illustrates the $\Theta(n \log n)$ algorithms of quicksort[1], merge sort, and heap sort. The second illustrates the $O(n^2)$ sorting algorithm insertion sort, and the $\Theta(n^2)$ algorithms bubble sort, selection sort, and rank sort.

The more efficient algorithms follow a nearly linear trend, while the less efficient algorithms are quadratic. Each algorithm has its own c constant factor, separating the curves. If one were to implement the algorithms in a different language (for example, assembly language[29]), the constant factors might vary, but the overall shape of the curves would remain the same.

Note the units on the axis of the figures – the run times of the $\Theta(n^2)$ start two orders of magnitude higher, and the problems considered are an order of magnitude smaller.

[1]Quicksort has a theoretical worst case behavior of $O(n^2)$, but when correctly implemented, the chances of this occuring are astronomically small.



(a) $O(n^2)$ sorting algorithms

(b) $O(n \log n)$ sorting algorithms.

Figure 3: Run times for two classes of sorting algorithms – an efficient $O(n \log n)$ set, and a less efficient $O(n^2)$ set.

The primary difference between the two sets of sorting algorithms is the computational complexity of the algorithms – they are all run on the same computer hardware, and implemented with the same language (C).

It should be obvious that the $\Theta(n \log n)$ algorithms are far superior to the $\Theta(n^2)$ approaches. Even if one were to accelerate an algorithm such as rank sort by a factor of one thousand, quick sort and the others would remain the best choice for even modest sized problems.

Only on small "toy" problems can the $\Theta(n^2)$ be faster than $\Theta(n \log n)$. By changing the relative processor speeds, programming languages, and so forth, the position of n_0 can be changed – but this is a losing proposition. The more efficient algorithm has an advantage that grows with the size of the problems encountered.

That the more computationally efficient algorithm will be faster is mathematically guaranteed. Knowing that algorithmic complexity is the key concern has allowed theoreticians to focus on efficient algorithms, free from the distractions of hardware implementation details. Theoreticians have not focused on parallel algorithms due to a lack of intellectual capacity, or because their thinking has been corrupted by a serial framework – parallelism is simply an orthogonal issue, of less importance than the overall computational complexity.

4 Willful Blindness

With a modicum of complexity theory in hand, we will now consider the challenges of parallel computing. Browsing a typical computer science textbook reveals only a handful of algorithms that can be accelerated linearly with additional processors. Amdahl's Law paints a grim picture: a software application based on classic algorithms can be expected to have non-trivial serial components – the performance gains possible with parallel resources are therefore severely limited.

Reconciling the mathematical upper bounds of Amdahl's Law and the enthusiasm for parallel machines reveals failings within the computer science research community that must be rectified. In a quest to improve the scalability of an application, many researchers are shifting towards algorithms that offer greater scalability – but at the expense of computational efficiency.

The issues we highlight in this paper are by no means new: in a 1991 paper from the supercomputing community, Dr. David H. Bailey described "twelve ways to fool the masses with parallel computing results[7]." Bailey's paper used humor to make a very serious point – that one could not always take a research paper at face value, and that reported results could be quite misleading.

Of the twelve ways noted by Bailey, the focus here is on the seventh, as it is particularly relevant to computer science. Bailey noted that it was possible to increase the parallelism of an application (and hence, the scalability), by using a less computationally efficient algorithm. As the previous section should have made clear, this is a profound error. In short, it is possible to develop approaches that *appear* to be beneficial, but are in fact profoundly wrong. In the works we mention, we wish to stress that it appears that errors are unintentional, and not deliberate attempts to mislead an audience.

4.1 A Specific Example: Shortest Path

For illustrating the use of an inefficient algorithm, we will consider a few published works related to finding the shortest path between a pair of points in a graph.

There are two main approaches to this problem. The first is Dijkstra's algorithm [11]. This algorithm is well known, and utilizes a common data structure, the priority queue (frequently implemented as an ordinary binary heap). The computational complexity of Dijkstra's algorithm os $O(E + V \log V)$, where E and V are the number of edges, respectively. If the graph is sparse (E is within a constant factor of V), one can simplify this to $O(n \log n)$, where n represents either the number of edges or vertices. The complexity of this algorithm is comparable to the efficient sorting algorithms mentioned above.

Dijkstra's classic approach is illustrated in Algorithm 3. This code is based on an example from the Cormen algorithms text [10]; we have integrated the edge relaxation code. Within the graph G, each edge has a cost (distance) w between vertices u and v. The distance to any vertex u is $d[u]$.

A second well-known approach to computing shortest paths is by Bellman and Ford[10], illustrated in Algorithm 4. The edge relaxation operation used by Dijkstra can be applied en masse, making a parallel (or vector based) approach easily applicable.

The computational complexity of Bellman-Ford on sparse graphs is $\Theta(n^2)$, similar to the less efficient sorting algorithms above.

Both algorithms are standard fare for any undergraduate computer science program, as is computational complexity. From theory alone, it should be obvious that for even modest sized graphs, Dijkstra's algorithm is faster and more energy efficient. Even if one were to accelerate the Bellman-Ford approach (by, for example, using massive parallelism), Dijkstra's algorithm would win out for large graphs. On small "toy" problems, a parallel approach could be faster – but this would have little practical value, because both algorithmic approaches would be remarkably fast.

From theory alone, it should be clear that it is pointless to pursue a parallel implementation of Bellman-Ford. From the work we now consider, it appears that this has been overlooked.

General Purpose Graphics Processors

Algorithm 3 Pseudocode for Dijkstra's shortest path algorithm. Q is a priority queue, with vertices ordered by distance from the start vertex s.

```
Initialize-Single-Source($G, s$)
$S \Leftarrow 0$
$Q \Leftarrow V[G]$
while $Q \neq 0$ do
    $u \Leftarrow$ Extract-Min($Q$)
    $S \Leftarrow S \cup u$
    for vertex $V \in Adj[u]$ do
        if $d[v] > d[u] + w(u,v)$ then
            $d[v] = d[u] + w(u,v)$
            Update-Queue($Q, v$)
        end if
    end for
end while
```

Algorithm 4 The Bellman-Ford algorithm, which can be made parallel easily, and which scales nearly linearly with the number of processors.

```
for $i = 0$ to the number of vertices do
    for each edge $(u,v)$ do
        if $d[v] > d[u] + w(u,v)$ then
            $d[v] = d[u] + w(u,v)$
        end if
    end for
end for
```

In [14], a general purpose graphics co-processor (GPGPU) was used to implement the Bellman-Ford algorithm. While tremendous gains compared to a serial Bellman-Ford implementation were achieved, the method is competitive with Dijkstra only on small graphs[13].

While perhaps a reasonable subject for illustrating the "CUDA" application programming interface, it is clearly a poor choice for actually implementing a working tool. This point was certainly less than clear from the original paper.

Custom Processors

A second instance that is more puzzling is a doctoral dissertation in which the author designed custom circuitry to compute shortest paths[19].

The underlying approach was a hardware implementation of the Bellman-Ford algorithm, for graphs that could be represented with a regular planar mesh structure. The "relax" operation was performed by scanning the mesh using a variety of different patterns (top to bottom, left to right, spiral, and so on).

In experimental results, it appeared that the custom processor was quite competitive with Dijkstra's algorithm for large graphs – until one notes that the implementation of Dijkstra's algorithm is incorrect.

Rather than using (for example) a binary heap to maintain the priority queue, the author instead applied heap sort to the data after each distance update. This resulted in an internal step of Dijkstra's algorithm to change from $O(\log n)$ to $O(n \log n)$, and gave a worst case run time of $O(n^2 \log n)$.

The implementation did not store all vertices in the priority queue from the beginning of processing, making the impact of this error less significant. For the problems considered, the incorrect implementation of Dijkstra's algorithm likely resulted in only about three orders of magnitude slow down (enabling the custom processor to appear competitive).

The observed run time growth for the implementation of Dijkstra's algorithm was roughly $O(n^{1.25})$, which one might expect would have piqued the authors curiosity; apparently it did not.

Reconfigurable Computing

Field programmable gate arrays (FPGAs) have become a popular substrate for reconfigurable computing efforts. The notion is that by enabling customized circuitry at low cost, processors targetted to specific applications become feasible.

In [43], the authors consider the implementation of Dijkstra's algorithm on an FPGA, and make the following comment.

> Since each step in Dijkstra's algorithm requires a number of operations proportional to $|N|$, and the steps are iterated $|N-1|$ times, the worst case computation is $O(|N|^2)$. Using priority queues the runtime of Dijkstra's algorithm is $O(|E| \lg |N|^2)$, which is an improvement over $O(|N|^2)$ for sparse networks. However, the space requirement increases and operations on priority queues are difficult to implement in reconfigurable logic, and for these reasons priority queues have not been dealt with in this paper.

Obviously, without a priority queue, Dijkstra's algorithm loses any advantage. The implementation in [43] appears to be similar to the Bellman-Ford algorithm, with the graphs being implemented with adjacency matrices (adding a further layer of complexity that would be unnecessary for sparse graphs).

"Speedups" reported in the paper range from a factor of 24 to a factor of nearly 68. As with the work of [19], this approach is in fact a massive slowdown.

Supercomputing

Another attempt to speed up shortest path computations was made by a team of researchers using a Cray supercomputer [35]. In some respects, the approach is a hybrid of the Dijkstra and Bellman-Ford algorithms. In the Dijkstra approach, the priority queue forms a serial bottleneck, restricting computation to consider only a single u vertex at any given time.

Their Δ-stepping algorithm performed speculative distance calculations, with alternating phases of computation and correction. Vertices are ordered into "buckets," and are processed in parallel. If the distance to any vertex in a bucket remains unchanged, then this speculative computation is effective, and the problem can be solved more swiftly. If the distance to a vertex is updated, the vertex must be reprocessed, and the additional work is wasted.

For a set of carefully constructed graphs, the approach provided performance gains – in realistic situations, however, one cannot predict graph structure. On experiments with graphs such as road maps, the approach was much slower than a conventional sequential approach[36]. The authors noted that obtaining scalable performance improvements for the shortest path problems remains an open challenge.

Summary of Shortest Path Errors

Three of works mentioned above make essentially the same error, using a computationally inefficient approach to the shortest path problem. In each of these cases, the simple serial implementation of Dijkstra's algorithm is faster for large graphs – and the performance losses (and power consumption increase) of the parallel implementations are literally unbounded.

In the fourth example with the Δ-stepping algorithm, the authors obtain an approach with comparable Big-O complexity, but performance gains are only seen for certain classes of carefully constructed graphs. On road maps, performance improves slightly with a few processors, and then degrades due to communication overhead. The higher constant factors of the approach prevent it from being competitive with the simple serial implementation of Dijkstra's algorithm.

There are other problematic flawed attempts to utilize parallel computing for this problem[41, 24], including a textbook that designed for university courses that advocates the parallel Bellman-Ford approach[32].

We expect this situation to be somewhat surprising. Dijkstra's algorithm is by no means obscure; it is a standard element of almost any undergraduate computer science education. Computational complexity theory forms the mathematical foundation upon which the research field is based. From Big-O complexity, it should be obvious that Dijkstra's algorithm is the better approach, and that the Bellman-Ford algorithm (even if accelerated with parallel resources) is not competitive with large graphs.

The authors of these works are not neophytes. They come from leading academic, industry, and government research groups. These works have appeared in competitive conferences and journals, edited textbooks, and a doctoral dissertation. Approaches that are orders of magnitude worse (in both run time and power consumption), and that require specialized hardware, software, and operating system support, are portrayed as a step forward. The authors themselves, as well as the reviewers, editors, and general audience, seem to have not noticed, or felt that the problems were not important enough to mention.

4.2 The Tip of the Iceberg

While we have focused in depth only on the shortest path problem, this is by no means the end of the trouble. That algorithmic efficiency should be critical likely goes against instinctive expectations – seeing hundreds or thousands of processors working together in harmony would seem to be the "best way." In reality, any rational software developer should use the most efficient available algorithm (unless the problem sizes are quite small).

Only rarely is the most efficient algorithm for a problem also massively parallel – and thus, only rarely can one employ large number of processors in a useful manner. If one moves away from requiring algorithmic efficiency, however, a whole world of parallelism opens up. One can find these algorithmic errors in a range of other areas.

- As part of a tutorial on GPGPU based programming, a $O(n^2)$ approach to rectangle overlap detection for lithography applications was presented[25]. There are a number of serial algorithmic approaches for this problem is based on computational geometry[40], with complexities of $O(n \log n)$. The fact that the parallel approach would be many times slower than the best serial algorithm is a point almost certainly missed by the majority of the audience.

- For the N-body problem encounted in many types of physics simulations, a simple $O(n^2)$ all-pairs approach that utilized GPGPUs was presented[18]. There is a well known $O(n \log n)$ serial approach by Barnes and Hut[8].
- Rather than calculating π using a standard series formulation, [39] suggests that Monte Carlo simulation (which is massively parallel) might be an alternate way to approach this and similar problems. It should be obvious that even with vast numbers of processors, the traditional serial approaches would obtain a more accurate estimate in less time.
- Rather than implementing the problematic serial Fibonacci computation shown in Algorithm 2, one can find a scalable, exponential time recursive approach in a reference manual for Intel's Threading Building Blocks (TBB)[42], and in the most recent edition of the well known algorithms textbook by Cormen, Leiserson, Rivest, and Stein[10]. The Cormen text makes abundantly clear what a bad idea this is, and stresses that it is used for illustrative purposes only. The TBB text simply shows an alternate "more efficient" approach in a sidebar.
- The parallel programming text by Lester[32] suggests the $O(n^2)$ rank sort algorithm as a viable approach, noting the following: *Since these calls are all executed in parallel, the total execution time of the whole parallel program is also O(n). This is superior to the fastest known sequential algorithms, which are all O(n* $\log$*n).*

4.2.1 The Impact on Amdahl's Law

To understand the attraction of inefficient algorithms for parallel applications, one must consider how this interacts with Amdahl's Law. Suppose we have an application where 10% of the work is serial in nature, and the remaining 90% is massively parallel. From a simple application of the law, it should be obvious that no more than a ten-fold performance gain can be had, no matter how many processors are available.

If, however, half of that 10% were involved in something such as computing shortest paths, or sorting numbers – switching to an inefficient algorithm makes the application appear much more scalable. The total amount of work to be performed explodes (from computational complexity, it should be clear that this expansion is bounded only by the problem size). The remaining 5% that is serial shrinks relative to the parallel work to be performed.

This is one step forward, and an unbounded number of steps backwards. If the audience is not familiar with complexity theory and the algorithms involved, however, they will likely miss how profound the error is.

4.2.2 Challenges Beyond Computational Complexity

Even when the most computationally efficient algorithm available is also massively parallel, one can still see gamesmanship. In the software development kit for the Cell processor[23], for example, there is a demonstration program which performs matrix multiplication. The output of the program would indicate that performance increases linearly with the number of processing elements – the floating point operation per second (FLOPs) double when going from one processor to two, and then double again with four. Upon inspection of the source code, however, it can be seen that only a portion of the run time is reported, and the reported results are peak performance, not the average over the entire execution. The time required to transfer data to and from the processing elements, and the initialization and release times for the processors, are omitted. When one includes this time, the performance gains are only 17% with a second processor, and 24% with all four. This is an example of Amdahl's Law; it should be obvious that further gains from more processors would be marginal at best.

Even when acknowledging Amdahl's Law, one can find instances where the impact is downplayed – for example, a parallel system can be compared against outdated or slow serial hardware. In [5], the authors discuss a hybrid system containing a single high-performance "fat" processor, and a large number of lower performance "thin" processors. With 10% of the workload being serial in nature, the system obtained a speedup of 16.7 – seemingly defying Amdahl's Law. If read carefully, one notes that the comparison is relative to a single low performance "thin" processor core. That a system with both faster processors, and more of them, should see substantial gains, is hardly surprising. If compared to the higher performance processor, it becomes obvious that Amdahl's Law remains in effect, with performance gains limited to the expected factor of ten.

In Bailey's original paper, twelve different ways of fooling the masses were described – while their use may have been curbed in supercomputing, they have flourished elsewhere.

4.3 Making Sense of the Errors

The research literature of computing, like any field, has its share of challenges – unintentional mistakes, and work that is poorly worded or researched.

It would be safe to assume that the errors mentioned above are not from intentional deception; rather, it appears that many researchers have simply lost touch with the importance of computational complexity, and assume that it would be possible to offset the loss of efficiency with greater parallelism. For example, one group suggests that following[4].

> If it is still important and does not yield to innovation in parallelism, that will be disappointing, but perhaps the right long-term solution is to change the algorithmic approach. In the era of multicore and manycore. Popular algorithms from the sequential computing era may fade in popularity. For example, if Huffman decoding proves to be embarrassingly sequential, perhaps we should use a different compression algorithm that is amenable to parallelism.

The algorithms in use today are the result of decades of careful work by theoreticians, seeking to minimize the computational complexity of each approach – and within those bounds, to minimize underlying constant factors. Integral to the nature of many efficient algorithms is serialization, through the reuse of intermediate values, and by enforcing structures to the data being operated on.

The computing community has been working for a half century to make parallel computing a practical reality, yet has been continually thwarted by the mathematics of computational complexity and Amdahl's Law. There are efficient massively parallel algorithms for only a handful of problems. One might hope to find more of these algorithms, but there seems little reason to expect that there is an abundance.

This situation is no doubt frustrating: it would seem that there would be nearly boundless performance gains through parallel computing, and that these gains are just slightly out of reach. One cannot underestimate the human element contributing to these errors – there is a great deal of pressure on researchers in both academia and industry to report "progress," and an environment where peer review clearly does not place sufficient emphasis on correctness. That ways to "fool the masses" have thrived should not be at all surprising.

Researchers in computing are not immune to the failings found in other fields[16, 15]; this has resulted in an abundance of errors, and a pernicious environment for real scientic advancement. Works that make the most optimistic assessments and the grandest claims – even if critically flawed – garner the lion's share of attention. Meanwhile, efforts that make more modest, but concrete contibutions, are overlooked. The errors we have noted above are not difficult to see – but it requires a willingness to look.

5 Why Massively Parallel Computing Will Fail (again)

While the computer industry seems to be pinning its hopes for sustained growth on massively parallel computing, these efforts will fail for a majority of application areas.

Computers are simply the embodiment of Turing's mathematical concept. Based on this framework, there is no fundamental difference between a parallel computer, and one that is serial. A serial computer can easily emulate an arbitrarily large number of parallel computers, with no more than a constant factor disadvantage. Just as changing the type of paper one might write on does not alter the foundations of mathematics, neither does switching from serial to parallel computing.

Software simply implements algorithms on the available hardware. Complexity theory shows clearly that *how* you solve a problem is far more important that *what* you solve the problem with. Different algorithms have different levels of innate parallelism, but few scale linearly with the number of processors. Just as changing the font or notation one might write with does not alter the foundations of mathematics, neither does switching from one programming language to another.

The algorithmic constraints apply to *all* software, not just portions that implement "classic" functions such as sorting and searching. No matter if it is simply coordinating the processing, linking subroutines together, handling user input, or providing output, there are serial constraints. Amdahl's observation, which has been backed up by literally decades of experimental evidence, is that the portion of work that is serial is non-trivial.

In all but a handful of applications, the upper bound on performance gains through parallelism is depressingly low.

These constraints, and their implications, are based on mathematics. Apart from shifting to an alternate universe where mathematics does not apply, there's little that can be done.

5.1 ... With A Few Exceptions

Those who are optimistic about the prospects of parallel computing will certainly point out areas where massive parallelism is useful.

These are areas where parallel computing has already been successful for many years, and fall almost exclusively into a framework outlined by Gustafson[17], who proposed alternate way to look at Amdahl's Law. Gustafson's suggestion was to look at the problem not as how to complete a task in as small of a time as possible, but rather, how to increase what can be accomplished in a fixed amount of time. If one can hold the serial portion of work to a constant, and increase the work done in parallel, massive scalability becomes feasible.

An obvious example of this would be computer graphics – the translation and rotation of triangles as part of a three-dimensional scene can be done in parallel, with the resolution and detail of images increasing with each generation of technology. Many video and image processing tasks have tremendous amounts of parallelism; graphics co-processors are designed to take advantage of this.

Other applications of massive parallelism that have been successful, and which will continue to benefit, include high volume servers for databases and web traffic, and a great many scientific applications. Any area where a large number of compute-intensive and independent applications must run simultaneously can also benefit. In some cases, response time is the primary objective, and one might trade computational efficiency for speed (provided the individual tasks are small enough).

One must be careful, however, to not assume that these applications comprise the entire universe of computing. Just as one should not infer the nature of an elephant from its tail, massive parallel scaling in graphics does not apply that it is useful everywhere.

The scaling envisioned by Gustafson clear does not apply to the shortest path problem, computation of the Fibonacci sequence, or a great many other things. In a recent debate[2], Gustafson noted that far more had been read into his work than he felt appropriate. To the amusement of the audience, he referred to a quotation from Ambrose Bierce, concluding that "even back in 1881, someone understood that there was something wrong with parallel processing."

5.2 Why The Problem Is Hard to See

Researchers in computing have grown accustomed to being surprised by performance gains, and have come to expect miracles on a regular basis. One would be hard pressed to find a projection more optimistic than Moore's Law – and yet, it has been realized, with semiconductor fabrication repeatedly moving past expected physical barriers.

Gains in semiconductor fabrication have sparked economic revolutions in a vast range of areas. Software has become an integral part of the modern world. New applications and industries seem to sprout out of thin air, far exceeding our wildest imaginations.

When one considers parallel computing, there is certainly a great deal of intuitive appeal. Much of what occurs in the physical world is massively parallel – the human brain is a remarkable, and obviously employs a great deal of parallelism.

What should be seen from a careful of study of algorithms, however, is that frequently the application of a series of coordinated operations can outperform brute force. Simply because nature employs massive parallelism, it does not imply that all problems are best solved in this manner.

The end result of this environment is that many seem to have a nearly unshakable confidence in the future of parallel computing, making them willing to overlook the fact that in many instances, the parallel solution is orders of magnitude worse than a reasonable serial approach.

All of this is missed due to simple human fallibility. The popular opinion is that parallel computing is the future – and few are willing to go against the grain. Further, research funding, publication opportunities, and career advancement are tied to winning the support and approval of the community as a whole – going against the grain can be a disasterous career move.

5.3 What To Do About It

Exactly what the research community should do about the apparent end to serial performance gains will no doubt be a subject of debate. There are many calls to revise the education of computer scientists to focus on parallelism. Clearly, however, the current system is failing to communicate the importance of computational complexity – if there is a curriculum change, we would suggest that there needs to be a greater emphasis on algorithms and the mathematical foundations of computing.

In particular, we would suggest the following.

- Emphasis should be placed on educating computer scientists on algorithms and complexity theory. There are catastrophic errors being made in parallel computing, which undermine the credibility of the entire area.
- Reviewers of technical papers and funding proposals should keep a careful watch for Bailey's "twelve ways to fool the masses." Promotion of incorrect work comes at the expense of correct work, and creates a pernicious environment.
- The research community should embrace real gains in parallel computing – which may be small. For example, one might accelerate quick sort by at most a factor of ten[31]; this is far more useful than the limitless scalability of rank sort. Similarly, even modest gains on Dijkstra's algorithm[34] have more benefit that massive speedup of the Bellman-Ford algorithm.
- Serial performance gains should remain a focus area. In his 1967 talk, Amdahl argued that the community should "keep the faith" in serial performance gains, and not abandon the approach because progress had become difficult. Further

progress will be far from easy[3, 28], but is essential for sustained performance gains.

- Considerable effort should be focused on minimizing serial bottlenecks, by accelerating the best available algorithms[33]. Chipping away at the "hard serial bottlenecks" raises the upper bound placed by Amdahl's Law.

Across the broad landscape of computing, there are few opportunities for massive parallel scaling – but an abundance of work that can be done on the small scale. For example, one study[9] of a broad range of applications have found that roughly 25% of loops are massively parallel – similar in nature to Algorithm 1. For this part of an application, additional processors provides some gains.

Ultimately, we must be realistic in expectations. Isolated cases of massive parallelism should not be treated as typical. Successfully harnessing thousands of processors with an inefficient algorithm does nothing to improve real performance. Acceleration through parallel resources will only carry the industry forward for a few years at best.

6 Conclusion

Two decades ago, there was a "golden age" in supercomputing, but much of the enthusiam was fueled by inflated expectations[7]. The field imploded[12], and has been slow to recover. Mainstream computing is now entering a "golden age" of parallelism, also fueled by inflated expectations. What outcome should we expect?

The computing industry is at a turning point, as serial performance gains appear to have ended. While we might expect semiconductor costs to continue to decline, chips will no longer be faster. Small scale parallel computing is a reasonable objective, as is acceleration of key elements of serial bottlenecks. Beyond a handful of processors, however, Amdahl's Law comes into play. Large scale parallel computing will remain limited to a handful of naturally parallel applications.

It is remarkably easy to "fool the masses," and give the appearance of improvement; the research community as a whole has failed to perform effective peer review to weed out fact from fiction.

While the skepticism we hold here may go against popular opinon, it is not an isolated position. Amdahl made his thoughts clear more than four decades ago, and has remained correct despite the best efforts of the research community. One of the most noted researchers in algorithms, Prof. Donald E. Knuth, recently made the following comment[30].

> 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.

Rather than repeating the mistakes of the previous decades, it is time to acknowledge the elephant in the room. There is no magic bullet that will make massive parallelism a universal solution. Dealing openly with the end of serial performance gains will be neither easy or enjoyable, but it must be done.

References

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

[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. ACM SIGDA member meeting at ICCAD, November 2007.

[3] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI throttling. In *Proc. Int. Symp. on Computer Architecture*, pages 298–309, 2005.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view form Berkeley. Technical report, UC Berkeley, 2006.

[5] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, pages 56–67, 2009.

[6] W. F. Atchison, S. D. Conte, J. W. Hamblen, T. E. Hull, T. A. Keenan, W. B. Kehl, E. J. McCluskey, S. O. Navarro, W. C. Rheinholdt, E. J. Schweppe, W. Viavant, and Jr. D. M Young. Curriculum 68: Recommendations for academic programs in computer science: a report of the ACM curriculum committee on computer science. *Communications of the ACM*, 11(3):151–197, 1968.

[7] D. H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, pages 54–55, August 1991.

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

[9] P. Borensztejn, J. Labarta, and C. Barrado. Measures of parallelism at compile time. In *Proc. HPC*, pages 253–258, 1992.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithm, 3rd Edition*. MIT Press, 2009.

[11] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[12] B. Furht. Parallel computing: Glory and collapse. *IEEE Computer*, 27(11):74–75, 1994.

[13] M. Garland. personal communication, 2008.

[14] M. Garland. Sparse matrix computations on manycore GPU's. In *Proc. Design Automation Conf*, pages 2–6, 2008.

[15] D. Goodstein. *On Fact and Fraud: Cautionary Tales from the Front Lines of Science*. Princeton University Press, 2010.

[16] J. Grant. *Corrupted Science: Fraud, ideology and politics in science*. AAPPL, 2006.

[17] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(3):532–533, 1988.

[18] T. Hamada and T Iitaka. The Chamomile Scheme: an optimized algorithm for N-body simulations on programmable graphics processing units. http://arxiv.org/abs/astro-ph/0703100v1, March 2007.

[19] P. M Hansen. *Coprocessor Architectures for VLSI*. PhD thesis, Computer Science Department, University of Califoria at Berkeley, Berkeley, CA, USA, 1988.

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

[21] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition edition, 2008.

[22] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[23] IBM DeveloperWorks. Cell broadband engine resource center. http://www.ibm.com/developerworks/power/cell/documents.html.

[24] M.Y. I. Idris, S. A. Bakar, E. M. Tamil, Z. Razak, and N. M. Noor. A design of high-speed shortest path coprocessor. *MASAUM Journal of Basic and Applied Sciences*, 3:531–536, 1.

[25] D. A. Jamsek. Designing and optimizing compute kernels on NVIDIA GPUs. In *Proc. Asia South Pacific Design Automation Conf.*, pages 224–229, 2009.

[26] A. K. Jones and P. Schwarz. Experience using multiprocessor systems – a status report. *Computing Surveys*, 12(2):121–165, 1980.

[27] P. H. Enslow Jr. Multiprocessor organization – a survey. *Computing Surveys*, 9(1):103–129, 1977.

[28] U. Karpuzcu, B. Greskamp, and J. Torrellas. The bubblewrap many-core: Popping cores for sequential acceleration. In *Proc. International Symposium on Microarchitecture*, December 2009.

[29] D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1997.

[30] D. E. Knuth and A. Binstock. Interview with Donald Knuth. InformIT.com http://www.informit.com/articles/article.aspx?p=1193856.

[31] C. E. Leiserson. The Cilk++ Concurrency Platform. In *Proc. Design Automation Conf*, 2009.

[32] B. P. Lester. *The Art of Parallel Programming, 2nd edition*. 1st World Publishing, 2006.

[33] J. Loew, J. Elwell, D. Ponomarev, and P. H. Madden. A co-processor approach for accelerating data-structure intensive algorithms. In *IEEE Int'l Conference on Computer Design*, 2010.

[34] J. Loew, D. Ponomarev, and P. H. Madden. Customized architectures for faster route finding in GPS-based navigation systems. In *IEEE Symposium on Application Specific Processors*, 2010.

[35] K. Madduri, D. Bader, J Berry, and J. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[36] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, chapter Parallel Shortest Path Algorithms for Solving Large-Scale Instances. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 2009.

[37] R. Miller. The status of parallel programming education. *IEEE Computer*, 27(8):40–43, August 1994.

[38] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, April 1965.

[39] D. A. Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–53, 2010.

[40] C. Preparata and M. I. Shamos. *Computational Geometry*. Springer, 1985.

[41] T. K. Priya, P. R. Kumar, and K. Sridharan. A hardware-efficient scheme and FPGA realization for computation of single pair shortest path for a mobile automaton. *Microprocessors and Microsystems*, 30(413–424), 2006.

[42] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[43] M Tommiska and J Skytta. Dijkstra's shortest path routing algorithm in reconfigurable hardware. In *Proc. 11th Conference on Field-Programmable Logic and Applications*, pages 653–657, 2001.

[44] A. M. Turing. On computable numbers, with an application to the *entscheidungsproblem*. *Proc. London Math. Society*, 2(42):544–546, 1936.