Mathematical Limits of Parallel Computation for Embedded Systems

Jason Loew Jesse Elwell Dmitry Ponomarev Patrick H. Madden
SUNY Binghamton Computer Science Department

Abstract– Embedded systems are designed to perform a specific set of tasks, and are frequently found in mobile, power-constrained environments. There is growing interest in the use of parallel computation as a means to increase performance while reducing power consumption. In this paper, we highlight fundamental limits to what can and cannot be improved by parallel resources. Many of these limitations are easily overlooked, resulting in the design of systems that, rather than improving over prior work, are in fact orders of magnitude worse.

I. Introduction

Embedded systems are pervasive; from cell phones and MP3 players to implanted medical devices, one can find dedicated computating systems everywhere. At the heart of any computing system is a processing core – an implementation of a Von Neumann architecture[13]. For many years, performance gains have been made through more sophisticated architectures, and increasing serial clock rates – but progress in this manner has dramatically slowed. High frequency devices have timing constraints that are difficult to meet without a great deal of power consumption. Even if the timing constraints can be met, the devices may not operate long enough on battery power, or dissipate too much heat. Architecture enhancements such as out-of-order execution and branch prediction now provide diminishing performance gains.

Many designs now incorporate parallel computation as a means to continue progress. At a first glance, this is an appealing proposition. One might cut the effective clock rate of a processor in half, but obtain a power reduction that is significantly greater than this. By fabricating two processors, one would in theory regain the performance of the original, but with much lower total power consumption. Modern semiconductor fabrication offers an abundance of transistors, allowing a single die to contain a great many processors.

There is now great enthusiasm for massively parallel designs, and in some instances, they can be effective. In this paper, we highlight how and when this approach is effective – and also where it is not.

II. An Illustrative Example

To provide a specific example to highlight how parallel computation can help an embedded system, as well as where it is of no value at all, we will focus on a GPS-based navigation device. This might be integrated into a vehicle guidance system, as a portable dedicated navigator, or within a modern cell phone.

Major components of such a system are illustrated in Figure 1. Key functions include determining exact locations (typically through GPS), obtaining updated route congestion information (through Wi-Fi or cellular data), route planning, route display, and user interaction (by voice control, for example).

For many of these tasks, special purpose hardware is clearly appropriate. Radio and GPS signals could in theory be processed by a general purpose compute core, but this would be horrifically inefficient – hardware solutions offer much greater performance, with much lower power consumption. By contrast, basic route display routines require limited graphics processing; this could be accomplished either with a general purpose processor, or with graphics acceleration hardware. If turn-by-turn directions are provided through voice synthesis, the processing required for this is quite modest; voice recognition by contrast might be better suited for specialized DSP-based hardware.

The route planning itself, however, poses an interesting problem, and we will use this to highlight the limitations of parallel hardware. The roads available can be easily modeled with a graph; at any given moment, one might need to compute the shortest or lowest cost path to a desired destination. Having this path known is essential for providing turn-by-turn guidance; changing traffic conditions will alter the best path in real time.

There are two main approaches for determining the path: Dijkstra's algorithm[7], and the Bellman-Ford algorithm[6]. Much of the paper will address why, even though the Bellman-Ford algorithm can be made massively parallel, Dijkstra's algorithm is the preferred solution.

III. Motivation for Parallelism

To be clear on the motivations behind parallel architectures, we will specify our objective as one of obtaining the desired performance (in terms of speed or power consumption) with a minimum of cost.

Processor architectures have evolved considerably over the years, shifting from very simple in-order execution into modern designs that employ out of order execution, sophisticated branch prediction, speculative execution, and multiple layers of cache hierarchy. Each advance in architecture has incurred a cost in terms of transistor count, as well as increasing power consumption. The expanding size and complexity of proces-

978-1-4244-7516-2/11/$26.00 ©2011 IEEE

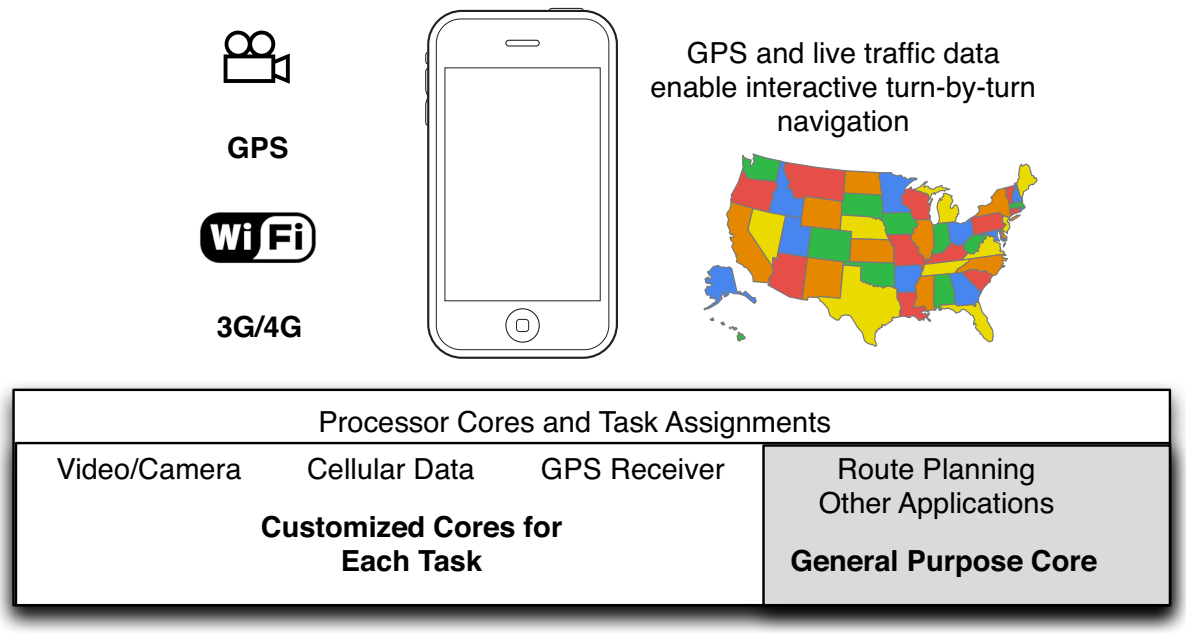


Fig. 1. Modern cell phones and personal navigation devices integrate a variety of functions into a single embedded system. Some functions are best handled by special-purpose processing cores; others can share general purpose computing resources.

sor designs has made increasing clock frequency a great challenge.

To model the tradeoff between design complexity and performance, Hill[14] proposed a simple "square root" metric. Specifically, $perf(r)$ represents the effective performance given r resources, with $perf(r) = sqrt(r)$ as a straw-man formulation. Doubling performance would require a factor of four increase in resources (loosely defined as either power, area, or both).

It should be obvious that if the work to be performed can be distributed across two small processors, this solution is much better than a single large processor. Two separate applications could easily utilize two different processor cores. The difficulty, however, is that not all tasks can be distributed across multiple processors. Thus, it is important to distinguish between tasks that can utilize parallelism easily, and more difficult cases.

There is **easy parallelism** if there are two or more completely unrelated functions that must be performed simultaneously. As described in our illustrative example, handling radio communications is best done on a dedicated processor – specialized hardware provides a significant performance improvement, and the simpler special-purpose design benefits from Hill's architecture metrics. The radio functions have only minor interaction with the rest of the system, and can operate essentially independently.

There can also be **easy serialization** if there are two or more completely independent non-simultaneous functions. It is possible to schedule each function onto a single general purpose processing core, and because they do not overlap in time, there is no competition. For the purposes of a navigation system, the path to follow must be determined first – afterwards, this information could be presented to the user. In our simple example, one might use the same processor to compute a route, and then to perform speech synthesis or update a visual display.

IV. Mathematical Limits of Parallel Computing

Interest in parallel computing is hardly new. The 1968 ACM Computing Curriculum[3], for example, lists multiprocessing and multiprogramming as key topics for research. A survey in the mid 1990's[23] 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.

The underlying issues that have kept parallel computing out of general purpose systems also apply to embedded systems.

A. Amdahl's Law

The most significant barrier to the use of high degrees of parallel computation is Amdahl's Law[1]. The fundamental observation, made more than four decades ago, is that while some sections are 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.

A simple example can be constructed, where one assumes that a portion p of an application can be accerated with parallel

computing. The remainder of the application $s = 1 - p$ is serial in nature. 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, speedups of more than a factor of two or three are uncommon.

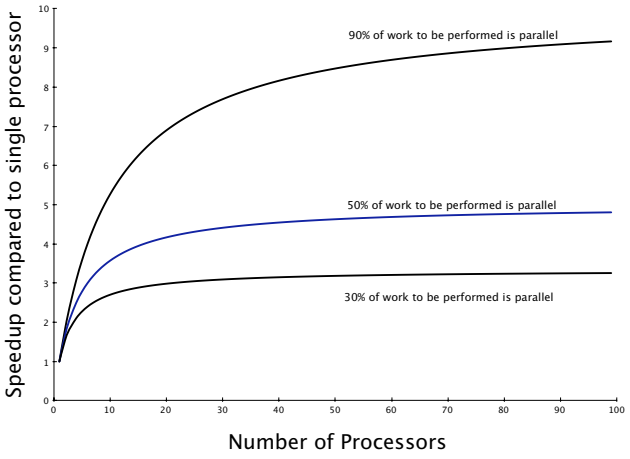


Fig. 2. An illustration of Amdahl's Law. Applications typically contain code segments that are parallel, and also segments that are serial. The serial portions place an upper bound on possible performance gains.

This is illustrated in Figure 2. 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[17, 6]).

B. Mathematics of Computation

Amdahl's Law is not an artifact of poor programmer ingenuity, or primative computer architectures. One can observe the simple fact that the amount of parallelism possible is a function of the *algorithm itself*, and not the hardware or software environment. To illustrate this point, consider the code samples shown in Algorithms 1 and 2. In the first, vector addition, there is an obvious way to exploit parallelism, and a large number of processors can be deployed efficiently. In the second example, a dynamic programming approach to computing the Fibonacci sequence, there is almost no parallelism that can be exploited.

We note that there are many different possible algorithms for computing the Fibonacci sequence, some that are more efficient than the method shown in Algorithm 2. We use this example as it clearly shows the dependence between loop iterations.

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

The point we wish to make is that opportunities for parallel speedup depend on the algorithms themselves. While multiple processors and a suitable programming environment are obviously necessary for parallelism, *they are not sufficient.*

C. No Substitute for the Right Algorithm

Given that the amount of parallelism available is a function of the algorithm itself, the tempting solution might be to simply select parallel algorithms. Leading research groups have proposed exactly this; for example, Asanovic[2] suggests the following:

> 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 sequen-

tial, perhaps we should use a different compression algorithm that is amenable to parallelism.

The error that can be easily made, however, is neglecting computational complexity. Pioneering work by Hartmanis and Stearns[12] showed that computational efficiency in the most important consideration when selecting an algorithm. To illustrate this, Figure 3 shows the growth of two algorithmic functions. It is possible (but not guaranteed) that a function with a higher growth rate can be faster for problems smaller than some value n_0; for large values, the most efficient algorithm will have the best run time (as well as having the least power consumption).

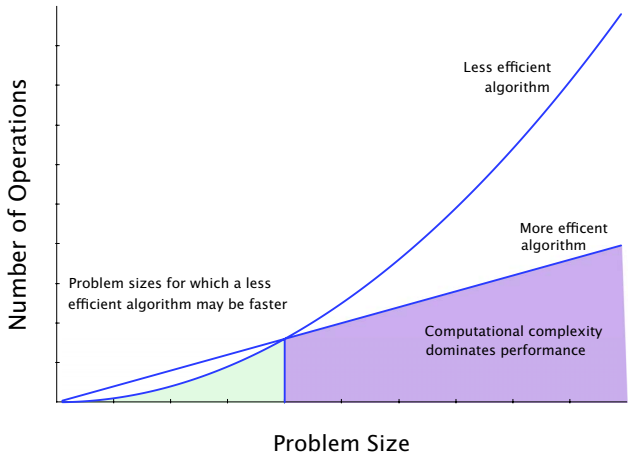


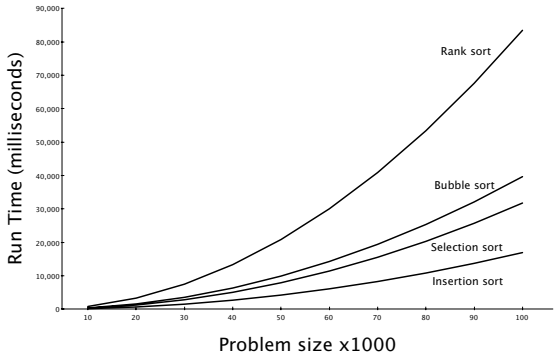
Fig. 3. The theoretical implications of computational complexity; if two algorithms with different complexities are implemented, the one with the lowest growth rate will be fastest for problems larger than some size n_0, regardless of the constant factor impact of processor speeds.

As a specific example, Figures 4(a) and 4(b) 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.

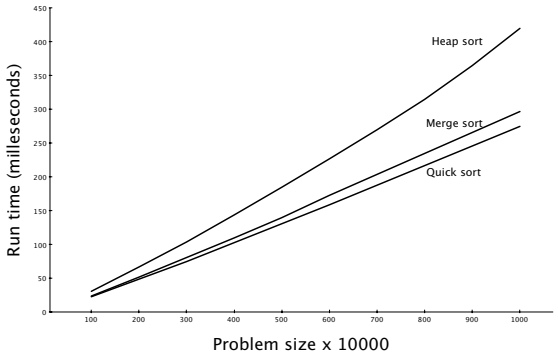
The more efficient algorithms follow a nearly linear trend, while the less efficient algorithms are quadratic. Each algorithm has its own c constant factor, separating the curves. If one were to implement the algorithms in a different language (for example, assembly language[16]), the constant factors 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. 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).

[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.

Fig. 4. Run times for two classes of sorting algorithms – an efficient $O(n\log n)$ set, and a less efficient $O(n^2)$ set.

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.

V. Impact on Embedded Navigation Systems

With a bit of background established, we return our focus to the problem of route planning. One can assume that the driver

of a vehicle will be able to receive live traffic updates via celluar or Wi-Fi, with a desired route changing periodically. There is a limited amount of time available to compute the best route to the destination, with some of that time being taken by the need for potential route changes to be announced, and the driver to respond to them.

There are two main approaches to the shortest path problem. The first is Dijkstra's algorithm [7]. 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 is $O(E + V\log V)$, where E and V are the number of edges and vertices, 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 [6]; 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 $d[u]$.

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 ⇐ 0
Q ⇐ V[G]
while Q ≠ 0 do
  u ⇐ Extract-Min(Q)
  S ⇐ S ∪ u
  for vertex V ∈ 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
```

A second well-known approach to computing shortest paths is by Bellman and Ford[6], 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.

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
```

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

Both algorithms are standard fare for any undergraduate computer science program, as is computational complexity. 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.

A. Implementation Errors

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.

In a doctoral dissertation, for example, the problem of vehicle routing was explicitly considered[11]. The objective of this work was to develop a method to plan routes through the San Francisco bay area, with the assumption that the area could be modeled with a regular grid graph. The author designed custom circuitry for this, and a chip was fabricated for experimentation.

The underlying approach was a hardware implementation of the Bellman-Ford algorithm. 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 presented[11], 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.

More recently, Garland[9] proposed using a general purpose graphics co-processor (GPGPU) 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[8].

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.

Field programmable gate arrays (FPGAs) have become a popular substrate for reconfigurable computing efforts. The no-

tion is that by enabling customized circuitry at low cost, processors targetted to specific applications become feasible.

In [25], 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 [25] 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 [11], this approach is in fact a massive slowdown.

B. Fooling the Masses

Algorithmic errors of the type noted in the previous subsection are not new. In 1991, Bailey[4] identified "twelve ways to fool the masses with parallel computing results" in supercomputing. The use of inefficient algorithms, reporting only portions of run times, and a variety of other tricks, resulted in a great deal of deceptive or misleading claims in the literature. These issues are still problematic[5].

C. No Acceleration With Supercomputing

The most dramatic performance gains through parallel computing have been obtained in "high performance" scientific applications (for example, [10]). There is a presumption that those involved in supercomputing have expertise, as well as hardware and software tools, that enable these performance gains. By applying this expertise, performance barriers might be in theory be removed; for the problem of shortest path computation, this does not seem to be the case.

The shortest path problem was carefully considered by a team of researchers using a Cray supercomputer[21]. In some respects, their approach was 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.

The authors proposed a Δ-stepping algorithm which performed speculative distance calculations, with alternating phases of computation and correction. Vertices were ordered into "buckets," and were processed in parallel. If the distance to any vertex in a bucket remained unchanged, then this speculative computation was effective, and the problem could be solved more swiftly. If the distance to a vertex was updated, the vertex needed to be reprocessed, and the additional work was 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[22]. The authors noted that obtaining scalable performance improvements for the shortest path problems remains an open challenge.

D. No Large-Degree Speedups

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 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[24, 15], including a textbook that designed for university courses that advocates the parallel Bellman-Ford approach[18].

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.

VI. Strategies for Hard Serial Bottlenecks

When faced with computation tasks where there is no computationally efficient, massively parallel algorithm, there are few good options. Selecting an inefficient parallel algorithm is a catastrophic mistake.

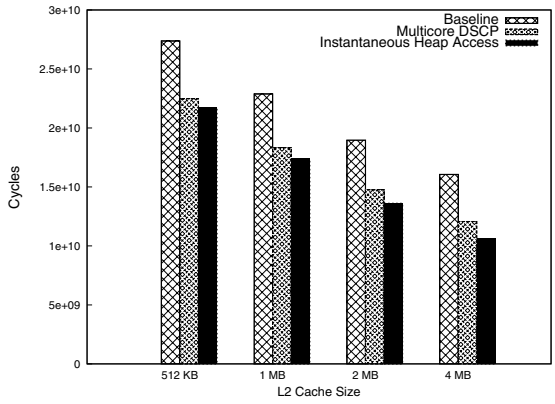
The only reasonable approach is to attempt to minimize the serial component of the application to the greatest extent possible, and to accept that performance gains (and the utility of multiple cores) are limited.

A great many efficient algorithms depend on a set of common data structures – heaps, priority queues, trees, and so on. To address the challenge of "hard serial bottlenecks," our recent research effort has focused on obtaining small scale parallel acceleration by distributing the work required for these data structures to co-procssors[19].

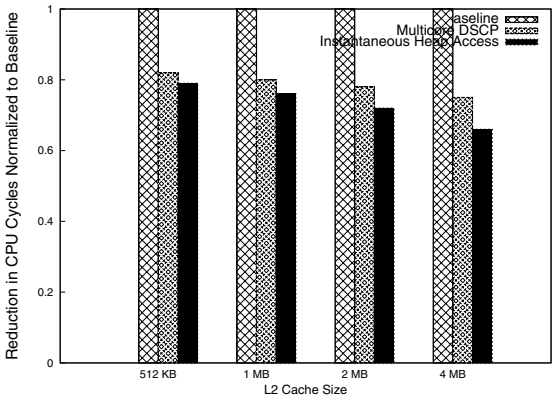
This "data structure co-processing (DSCP)" approach is applicable to many data structures: binary search trees, red-black trees, heaps of various types all utilize basic operations of *insert*, *delete*, and a few types of queries, to maintain collections of data[6]. These operations typically take hundreds or even thousands of instructions (depending on the data structure size) to complete – large enough that it is worthwhile to look for ways to overlap processing, but small enough that low-overhead implementations are essential.

Using the DSCP approach, Dijkstra's algorithm can be accelerated (although by not much more than about 25% for graphs based on road maps)[20]. Cycle accurate simulations, using a primary processor to handle the main loop of Dijkstra's algorithm, and a secondary processor to manage the priority queue, provides some gains. Results of these experiments are shown in Figure 5.

The slight gains for this approach might not appear as impressive as the massive scalability of the parallel Bellman-Ford approach – but we would hope that it would be clear than in practice, even slight gains on Dijkstra's algorithm are much more valuable.



(a) In CPU Cycles



(b) Normalized to the Baseline Performance

Fig. 5. Execution Time of the Full USA Benchmark for Different L2 Cache Sizes.

VII. Conclusion: Smart Design for Embedded Systems

Designers of parallel systems must accept Amdahl's Law, and deal with it squarely. The following points should make the challenge clear.

- The amount of parallelism available is a function of an algorithm itself. One cannot inject additional parallelism into an algorithm through either hardware or software modifications.
- The computational complexity of an algorithm is the most important consideration. It may be the case that there is a massively parallel algorithm (such as Bellman-Ford) for a problem; if it does not match the computational complexity of the best possible serial approach, it will be both slower, and less power efficient.
- The application and user requirements define which algorithms can be used. If the application contains problems that are best solved by an inherently serial algorithm, these form the basis of a "serial component" in Amdahl's mathematical framework.
- The portion of work that is serial in nature will quickly dominate the run time of any massively parallel approach.

We should note that it is remarkably easy to mask the effects of Amdahl's Law, and to obtain inflated performance results. As noted, Bailey's "twelve ways to fool the masses" still occur.

In the prior sections, we have attempted to make clear where parallel computing is useful, as well as where it is not.

If tasks can be distributed easily, i.e., there are a number of essentially different and independent functions, multiple cores is an easy solution. In embedded systems, there would be little reason to not use a customized core for processing radio signals, for example.

For tasks where it is essential to accelerate performance, the nature of the algorithms comes into play. Graphics, for example, is naturally parallel – multiple cores is a logical solution. The shortest path problem, by contrast, lacks a computationally efficient highly parallel solution.

We summarize our observations with the following.

A. Exploit Available Parallelism

First, if the application can be broken down into independent tasks, parallel computation is an extremely effective technique. For our example with a navigation system, the cores which handle radio communications, display, and route processing, have

very little interaction – it would be a mistake *not* do utilize separate cores.

Further, for some tasks there are efficient, massively parallel algorithms. For these, massively parallel architectures are a valid approach. The rendering of graphics, and image processing in general, are efficiently handled with parallelism.

B. Use Efficient Algorithms

Perhaps the most important issue to grasp in the design of an embedded system is the paramount importance of the underlying algorithms. Many circuit designers, having focused on the underlying architecture and circuit issues, have only a limited knowledge of computer science algorithms and complexity theory. This leads to the great danger that an inefficient algorithm may be implemented simply because it is "naturally parallel."

This has occurred repeatedly in the literature, and in designs that have been fabricated. This sort of error is deadly to any commercial product – it hands to a competitor a performance advantage that is arbitrarily large.

For algorithms that are not massively parallel, it is possible to extract small amounts of acceleration by distributing portions of work. In the shortest path problem, speedups of 25% are possible. In sorting, one might expect as much as a factor of 10[17]. In general, however, gains will be quite modest.

C. Have Realistic Expectations

Ultimately, it is important to have realistic expectations, based on the nature of the problem under consideration. The high scalability of certain tasks in supercomputing, or in consumer graphics, do not necessarily apply to other areas. Clever interconnect architectures and compilers do not alter this – one can only leverage the parallelism inherent in an algorithm.

One should also examine claims of high degrees of speedup carefully. It is remarkably easy to "fool the masses." With the intense pressure to increase performance, many researchers are all to willing to believe "good news," even when that news is too good to be true.

Amdahl's Law paints a grim picture for massive parallelism. Prudent designers would do well to approach multi-core design with a great deal of caution, and to not fall victim to the hype.

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] 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.

[3] 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.

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

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

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

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

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

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

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

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

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

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

[14] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, July 2008.

[15] 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.

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.