Massively Parallel Processing: It's Déjà Vu All Over Again

Steven P. Levitan
Electrical and Computer Engineering
University of Pittsburgh
Pittsburgh, PA 15261
+1-412-648-9663
levitan@pitt.edu

Donald M. Chiarulli
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
+1-412-624-8839
don@cs.pitt.edu

ABSTRACT

In this paper we will identify those aspects of the concurrent computing landscape that have changed since the 1980's and how those changes might impact the efficacy of parallel computing as we move from single- to multi- to many- and to massive numbers of processing cores.

Categories and Subject Descriptors

C.1.4 [Parallel Architectures]. D.1.3 [Concurrent Programming]: Distributed programming, parallel programming. B.7.1 [Integrated Circuits]: Microprocessors, Very Large Scale Integration (VLSI).

General Terms

Algorithms, Performance, Design.

Keywords

Multicore, Massively Parallel Processing, Parallel Architectures and Algorithms.

1. INTRODUCTION

Based on the ideas of Amdahl's law [1], Gustafson's rule [9], Stone's performance metrics [16], Synder's corollary of modest potential [15], and Finnegan's approach [5], the promise of massively parallel processing (MPP) is only superseded by the hyperbole of the discussions to date. What is clear is that while some applications naturally benefit from parallel processing others simply do not. And, simple extrapolation of increased performance with increasing hardware is naïve and perhaps dangerous [2].

This means that, from the user perspective, we will not see significant qualitative changes in system performance by applying parallelism to current applications. Rather, all is not lost. What have changed over the 30 years of parallel computing history are the user's expectations. Rather than single algorithms, there are significant new application domains such as conversational interfaces, personal sensor networks, and various forms of intelligent assistants, that can be enabled by highly integrated and massively parallel hardware.

Over the next decade we must go beyond the use of threads for small scale parallelism, and process migration for medium scale systems to concurrent implementations of computationally intensive applications.

There are three obvious differences between the integrated multicore environments proposed for the next decade and the parallel supercomputers of the previous century. First, the hardware environment on-chip is different than the supercomputer environment in terms of communications latency, memory bandwidth, and I/O overhead relative to CPU processing power. Second, the applications themselves will not be the scientific supercomputer applications that were the focus of much research in the past. While many applications will be based on the signal, image, and graphic processing algorithms developed for parallel super-computers these new applications will be focused on meeting individual user's needs. Finally, these new systems will be general purpose personal computers. They will be running many applications in a multiprocessing environment that is likely to be on a mobile platform versus the dedicated applications that in the past ran on special purpose processors in "batch mode."

In the rest of this paper we introduce the rational for multicore processors, using concurrency rather than machine complexity or clock speed to increase performance. Then we present some of the problems with simple models for the performance gains predictions that have been used in the past and describe the software and hardware issues that are key to performance gains. Finally, we discuss the differences between multicomputer based parallel processing and the multicore environment.

2. THE DRIVE TO MULTICORE

The drive to multicore is propelled by the goal of increased performance and the "walls" single CPU architectures have encountered in terms of instruction level parallelism (ILP), clock frequency, and ultimately power. Architectural techniques to provide more performance have reached a point of diminishing returns and simply turning up the clock speed is no longer an option. Therefore, it makes sense to back-off from the high ILP performance, high clock speed, and deeply pipelined designs to more moderate, lower speed, lower voltage, and lower power CPU designs and utilize parallelism to achieve the performance gains we are seeking.

However, the performance gains of a multicore processor are predicated on the effective utilization of the cores to perform useful work. Therefore, it is interesting to examine the effectiveness of the potential parallelism for given applications and the factors that limit the efficiency of multiprocessors. Over the last several decades, the question of "how much speedup is possible?" has been examined for a large variety of architectures, algorithms, and fundamental assumptions about the kinds of programs that would be run in a parallel computing environment. As we show below, the differences in the assumptions made by researchers has been quite varied.

3. SPEEDUP ANALYSES

One way to analyze advantages of parallel execution is to measure the speedup over an equivalent serial implementation of the same algorithm: $Speedup = Time_{serial} / Time_{parallel}$. In 1967 Gene Amdahl noted that, if any part of the algorithm was "intrinsically serial" then that fraction would limit the overall performance of any parallel implementation [1]. To see this effect, we use the time to do the work on a serial machine with S as the fraction of that work that is intrinsically serial and P as the fraction that is parallelizable, and hypothesize that the speedup for a parallel machine would be: $Speedup = (S + P)/(S + P / N)$, with $P = 1 - S$. This gives: $Speedup = 1/(S + (1 - S)/N)$. This function is plotted in Figure 1 for values of S ranging from 10% to 90% and N from 1 to 1024. Even in the best case shown, for S = 10% and N = 1024 the speedup is only 10x. This pessimistic result comes from the fact that the serial fraction of the code cannot be sped up, no matter how many processors you can recruit to solve the problem.

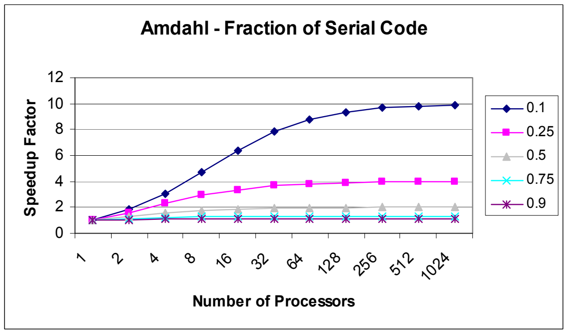

Figure 1: Speedup as a function of the number of processors, for a fixed problem size and various fractions of intrinsically serial code.

A different view has been taken by many researchers who claim that the real win for parallel processing is that it allows you to scale the problem size to larger instances, and then solve the larger problem quickly with a large number of processors. This interpretation was taken to extreme by D. Cohen in 1981 who reported "…in the 70s … scientists discovered… [that] problems with [high] time complexity can be solved in [little] time… using a number of processors which is a function of the problem size..." And further that "…the cost of VLSI processors decreases exponentially. … Hence, the application of an exponential number of processors does not cause any cost increase…" He attributed this result to Finnegan [5].

For our interpretation of Finnegan's rule we let the work grow with the number of processors while keeping the amount of serial work constant, this gives: $Speedup = (c + P)/(c + P / N)$ which is shown in Figure 2

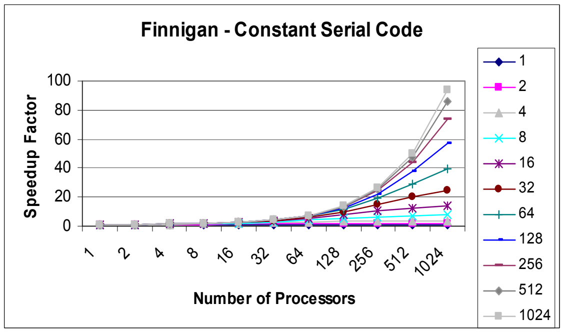

Figure 2: Speedup as function of the number of processors, for various data-set sizes given a fixed quantity of intrinsically serial code.

for a value of C=10. This gives a much more optimistic speedup of almost 100 for a 1024 processor architecture.

Another optimistic analysis was performed by Gustafson and Barsis [9]. In that work, they use the time to solve a large problem in a parallel environment, and then examine what fraction could not be run in parallel. From that information, they hypothesize the time the same problem would take in a serial environment. $Speedup = (S' + P' \cdot N)/(S' + P')$ where the prime indicates the fact that these are the relative serial and parallel fractions of the parallel code implementation. Thus, the hypothesized serial time is computed based on the data scale factor N. This gives a very optimistic: $Speedup = N + (1 - N) \cdot S'$, which is to say nearly linear speedup with problem and architecture size. To understand this relation, shown in Figure 3, one can consider an application running on a 1024 node parallel processor and taking 2 seconds. If half of that code ran as serial code and half as parallel code, then one could assume that a uniprocessor would take 1+1024 seconds to solve the same problem. Thus giving a speedup of 512x.

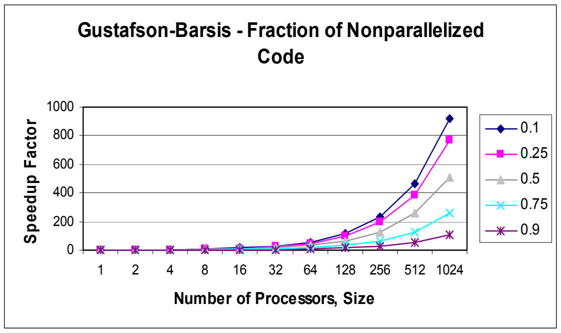

Figure 3: Speedup, in terms of the relative slowdown of a serial processor, as a function of the number of processors and data set size, for various fractions of intrinsically serial code.

Looking beyond the runtime for a fixed size problem, Snyder and others note that what matters is the amount of useful work that can be done in a fixed time as the number of processors increase [15]. Figure 4 shows that for algorithms with

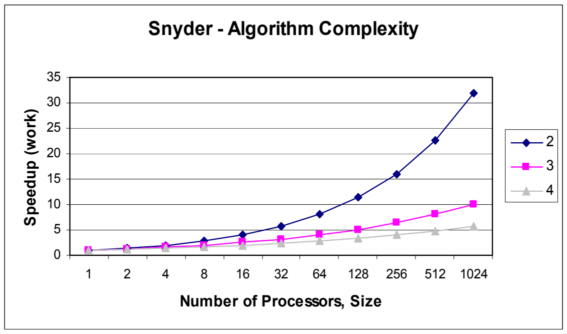

Figure 4: Speedup, in terms of work completed, as a function of the number of processors for various exponents, *x,* for polynomial time algorithms *O(N^x).*

polynomial complexity, O(N^x), the increase in the amount of work done grows as the inverse power (1/x). As shown in Figure 4, a 1024 node multicore machine solving an O(N^2) problem will only be able to complete 35x more work in the same time as a uniprocessor.

A further complexity in predicting runtime performance is due to the communications overhead of the cooperating processes or threads in many parallel applications. Stone looks at runtime in terms of both computation and communication costs [16]. He defines the ratio *R/C* as the ratio of runtime, *R,* to communication time, *C,* for each block of code that could be run in parallel.

Stone notes that using massive parallelism where short blocks of code communicate frequently can lead to very poor speedups if the communications take finite time. Given *M* tasks and *N* processors, and for simplicity it is assumed that all tasks need to communicate with each other, the speedup can be characterized as:

$$\frac{RM}{\left(\frac{RM}{N}+\frac{CM^2}{2}-\frac{CM^2}{2N}\right)} = \frac{N\,{}^{R}\!/\!{}_{C}}{{}^{R}\!/\!{}_{C}+\frac{M(N-1)}{2}}$$

Where we can see, on the left, the numerator is the total runtime of all the tasks on a serial processor and the denominator has the run time for the parallel tasks and the communications time for all tasks except those tasks that share processors. The re-ordered equation, on the right, shows that only if *R/C* is much larger than *MN* will the speedup grow linearly with *N*. This effect is shown in Figure 5, where for all the plots, *M,* the number of processes is kept constant at 2000.

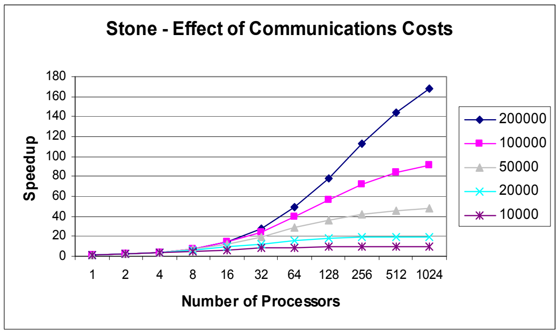

Figure 5: Speedup as a function of number of processors for various values of the ratio of runtime to communication time, *R/C,* for 2000 processes.

This ratio of computation to communication time is critical to achieving any significant speedups in parallel processors and therefore it has been the subject of research from both a hardware and a software perspective. First, in terms of computer architecture, the question is: How can we build interconnect systems that minimize the latency of communications between processors? And second, in terms of parallel algorithm design, the question is: How can we partition our problems such that we can minimize the communications between processes?

4. COMMUNICATIONS

4.1 Interconnection Networks

The design of system level interconnection (SLI) networks is one of the classic problems in computer architecture. The space of possible networks can be defined in several ways. One classification scheme is based on the four axes: operation mode (synchronous vs. asynchronous), control strategy (centralized vs. distributed), switching method (circuit vs. packet), and network topology [7]. Topologies can be classified as static or dynamic, single or multi-hop, directed graph, hierarchical, or multi-stage [14].

One key metric for the design of interconnection networks is their scaling behavior. For the desirable case of single hop fully-connected networks the space of network topologies span the range from pair-wise direct connections, which are limited in scale by their need for O(N^2) wires, to many-to-many bus-style interconnections, which are limited by their fan-in/fan-out loading that scales as O(N^2). Corresponding to those choices, the control and arbitration of these networks range from simple pair-wise handshaking to global arbitration of the shared resource.

Traditionally network designers have addressed these scaling issues by choosing to use multi-hop or hierarchical switched networks. For example, in a crossbar switch interconnection network O(N^2) switches replace the O(N^2) wires of a fully interconnected network, with bandwidth, routing, and control latency managed at the switch in the network core. To make this approach scalable, the switching architecture is broken down in various ways to create a network of smaller switches such that each message makes multiple hops within the network core. This is the most common architecture of modern network-on-chip (NOC) designs for providing interconnection between components at the system level [3].

Both the earlier work and newer analysis, such as [10] have shown that nearest-neighbor, mesh interconnection networks do not scale well for large systems due to the large diameter of nearest neighbor topologies. Rather, networks that have higher node degrees but lower diameters such as hyper-cubes, multi-grids, pyramids, or meshes-of-trees topologies have better performance for moderate increase in costs. In fact, even random networks turn out to be better than meshes for many applications [6].

A good deal of research has been done on mapping specific parallel algorithms onto specific interconnection networks [11][12]. While this exercise has highlighted the power and weaknesses of particular networks it has not converged on a single "universally best" network for all applications. Rather the communication graph of each application often determines the optimal embedding. These graphs come from the method used to decompose the original problem.

4.2 Parallel Programming Paradigms

From a high level perspective there are four classic methods for decomposing a problem for solution on a parallel computer. These are:

- Decomposition in "space" – Map some aspect of the data representation of the problem to individual processors. This can be straightforward such as putting a 2D weather map onto a mesh of processors or mapping nodes of a distributed data-structure such as a graph onto subsets of processors.
- Decomposition in "time" – Pipeline partial results either at a fine granularity, as in the vector unit of a single CPU, or across processing units in signal processing pipelines or systolic arrays.
- Decomposition by "function" – Assign tasks to processors based on specific capabilities of the heterogeneous processors, or to balance workloads.
- Decomposition by "instance" – Run multiple instances of problem, with different input conditions, on different processors.

While the last technique is trivial, it is probably the most common and effective use of parallel computing clusters. On the other hand, the first technique encompasses most of the research on parallel algorithm (and language) design.

One thing that is clear is that the same problem can be solved with algorithms that might or might not lend themselves to parallel speedups. The amount of "intrinsically serial code" in an implementation is not the same as any intrinsically serial part of the original problem.

What we have learned is that finding good parallel algorithms is not easy, and once those are found, mapping the algorithms onto an interconnection network to minimize the amount and cost of communications is also difficult. However, it is not clear that the success of multicore systems will simply be based on how well they run particular parallel algorithms.

We now turn to looking at three key differences between parallel processing in the past and the prospects for multicore architectures of the future.

5. WHAT IS DIFFERENT NOW?

5.1 The Hardware

The first question we consider is: "Does the change in size from macro-scale to chip-scale parallel computing really change anything?"

The obvious advantage of multicore platforms is the high level of integration. Processor cores are smaller, logic is faster, and individual threads run faster. But, as we showed above, none of this means anything if the interconnection networks cannot efficiently move the data between computational cores. Decreased size and increased scale give a two edged sword, on the one edge interconnection length is, on average, a function of the die size, while on the other edge increased logic density puts N^2 more resources at each incremental unit of distance away. This problem is exacerbated by the trend to keep clock rates fixed even as technology scales down.

What sort of interconnection networks can we design in this technology regime that will support efficient, low latency communications? While design paradigms have shifted in multicore environments, the same impediments that have historically blocked a general-purpose parallel architecture still exist. We are likely to continue to see more application driven architectures that attempt to balance the market size for the application with development costs. Algorithm development will precede these architectures, as new user applications emerge.

On the positive side, there is a significant paradigm shift that we can exploit in on-chip interconnection networks for multicore processors. Looking at the interconnection networks operating at the board and rack level of modern supercomputers, we typically find hierarchical switched networks that store and forward packets of data over shared physical links. These networks provide the low diameter requirements to keep latency low, but often are bandwidth limited due to cost constraints.

Many academic proposals for large-scale multicore chips are based on Network-on-Chip architectures (NOC's) that scale-up the switched hierarchical networks of supercomputers to chip-scale solutions. For these switched networks, one of the primary motivations is to efficiently share the physical links in the networks, which are a conserved resource in traditional supercomputer designs. This efficiency is gained at the price of end-to-end message latency associated with multiple store-and-forward operations as each message traverses the network.

On the other hand, slotted rings are common in current commercial multicore processors such as the Intel Larrabee [13] and IBM Cell Broadband Engine [8] where bandwidth is not so costly and network diameters are still moderate.

It is not clear that either design paradigm holds in highly integrated systems. In this environment, the number of physical interconnection is only related to the number of metal layers and shorter distances reduce the cost of delivering data across any single interconnection. This is not to say that multicore interconnection network would not include shared links and hierarchy. The key is to find a network management paradigm that can manage this sharing in a way that exploits the technology to minimize latency.

There are two design paradigms that seem capable of meeting this requirement. The first is a traditional switched approach paired with some sort of look-ahead mechanism to reduce the network setup time. The second is to make the network purely passive, for example a hierarchy of multiple fan-in/fan-out structures with all control operations moved to edges and reduced to a combination of bus/route selection and access arbitration [4].

5.2 The Software

The second question is really: "Have the user's needs changed?"

There is in computing a synergistic relationship between available technology and the user's needs and expectations. The classic modality is that new technology created new markets. However, today it's just as likely that it is user demands and expectations that are pushing the development of new technologies to meet these needs.

To understand how the answer to this question can drive the development of multicore parallel architectures, we need to make the connection between these new demands and the potential for new parallel system solutions that pair these applications with multicore parallel computing engines.

Without suggesting that we have a crystal ball about the "next big thing," there are certainly applications that seem to have intense computation requirements, whatever algorithms ultimately emerge as a solution, that are close and personal to end user. Spoken interfaces are an obvious application. However, more generally there is a great variety of end-user oriented applications that may emerge which range from cognitive prosthesis and quality-of-life enhancement for older individuals, to data-fusion and processing for personal "sensor networks", to wearable or embedded applications in our homes and transportation systems, and personal data miners that watch what we do and then suggest what we want.

The answer to the role of multicore architectures in the future lies in the relationship between the system architecture and the solutions that meet these new user demands.

5.3 The Application Domain

Rather than consider "what algorithms will we be running?" the last question is "What will these systems really be doing?"

The answer to that, we think, is that they will be interacting with individual users in their day-to-day lives. To that end, we should not be thinking about individual algorithms but rather higher level computational metaphors that reflect the need to work in the user's problem domain with constructs that directly support computational intensive applications. These domains include:

- Interpretation – reacting to commands by delegating the execution of abstract protocols, controlling physical and computational systems.
- Transformation – partitioning and translating different representations, performing data analysis, signal processing, and communication.
- Simulation – using cooperating processes to predict the behavior of physical, artificial, and social systems.
- Optimization and search – exploring state-spaces representations and processes with distributed resources.

Each of these domains, as well as others, provides a rich field of opportunities for using concurrency to solve problems more effectively, consuming less power, than a uniprocessor solution.

The difference between the problems that were solved by parallel processing systems of the past decades and the multicore processors of the future will be in their value to individual consumers.

6. ACKNOWLEDGMENTS

This work was supported, in part, by the National Science Foundation under grant CCF-0541150.

7. REFERENCES

[1] Amdahl, Gene, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967.

[2] Bailey, David H., "Misleading Performance Reporting in the Supercomputing Field," RNR Technical Report RNR-92-005, December 1, 1992. also in *Scientific Programming, vol. 1.*, no. 2 (Winter 1992), pg. 141–151.

[3] Benini, L. and De Micheli, G. ,"Networks on Chips: A New SoC Paradigm," *IEEE Computer*, Jan. 2002, pp. 70-78.

[4] Chiarulli, D.M., Levitan, S.P., Melhem, R.G., Teza, J.P., Gravenstreter, G., "Partitioned Optical Passive Star (POPS) Multiprocessor Interconnection Networks with Distributed Control, *IEEE Journal on Lightwave Technology, Vol. 14*, No. 7, pp. 1601-1612, July 1996.

[5] Cohen, D, "The VLSI approach to computational complexity," in *CMU Conference on VLSI Systems and Computations*, Kung, Sproul Steele ed. Computer Science Press, Rockville, MD., pp. 124-125 1981

[6] Falman, S.E., "The hashnet interconnection scheme," Carnegie-Mellon University, Dept. of Computer Science (CMU-CS-80-125) 1980.

[7] Feng, T.Y, "A Survey of Interconnection Networks," *IEEE Computer*, December, 1981, pp 12-27.

[8] Gschwind, M., Hofstee, H. P., Flachs, B., Hopkins, M., Watanabe, Y., and Yamazaki, T. 2006, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro* 26, 2 (Mar. 2006), 10-24.

[9] Gustafson, John L., "Reevaluating Amdahl's Law," *Communications of the ACM 31*(5), 1988. pp. 532-533.

[10] Kim, J., Balfour, J., and Dally, W., "Flattened butterfly topology for on-chip networks," *IEEE Computer Architecture Letters,* vol. 6, 2007.

[11] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*, Morgan Kaufman, San Mateo, CA, 1992.

[12] Levitan, S.P., "Measuring Communication Structures in Parallel Architectures and Algorithms," (in) *The Characteristics of Parallel Algorithms*, L. Jamieson, D. Gannon, R. Douglass, Eds., Cambridge, MA, MIT Press, 1987, pp. 101-137.

[13] Seiler, L., Carmean D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P., "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics,* 27, 3, 2008.

[14] Siegel, H.J. *Interconnection Networks for Large-scale Parallel Processing: Theory and Case Studies*, McGraw-Hill, 1990.

[15] Snyder, Lawrence, "Type Architectures Shared Memory and the Corollary of Modest Potential," Ann. Rev. Comput. Sci. 1986. 1:289-317.

[16] Stone, Harold, *High-Performance Computer Architecture*, Addison-Wesley, MA, 1987. (see chapters 1 and 6).