CTWatch QUARTERLY

ISSN 1555-9874 VOLUME 3 NUMBER 1 **FEBRUARY 2007**

# THE PROMISE AND PERILS OF THE COMING MULTICORE REVOLUTION AND ITS IMPACT

GUEST EDITOR
**JACK DONGARRA**

## INTRODUCTION



## FEATURE ARTICLES



**Available on-line** at http://www.ctwatch.org/quarterly/

CyberInfrastructure Partnership

Cyberinfrastructure Technology Watch
http://www.ctwatch.org/

CTWatch QUARTERLY

ISSN 1555-9874 VOLUME 3 NUMBER 1 **FEBRUARY 2007**

# THE PROMISE AND PERILS OF THE COMING MULTICORE REVOLUTION AND ITS IMPACT

GUEST EDITOR
**JACK DONGARRA**

# Introduction

Jack Dongarra
University of Tennessee
Oak Ridge National Laboratory

Over the past few years, the familiar idea that software packages can have a life of their own has been extended in a very interesting way. People are seriously exploring the view that we should think of software as developing in and depending on a kind of ecological system, a complex and dynamic web of other software, computing platforms, people and organizations.[1] To the more technical segment of the cyberinfrastructure community, this concept of a *software ecology* may seem to be just a metaphor, more suited to marketing than sound analysis and planning. Yet we all know that good metaphors are often essential when it comes to finding a productive way to think and talk about highly complex situations that are not well understood, but which none the less have to be confronted. The dramatic changes in computing discussed in this issue of *CTWatch Quarterly – The Promise and Perils of the Coming Multicore Revolution and Its Impacts* – represent an extreme case of just such a situation. We should therefore expect that the heuristic value of the software ecology model will be put to a pretty severe test.

[1] Messerschmitt, D. G., Szyperski, C. *Software ecosystem : understanding an indispensable technology and industry*. Cambridge, MA: MIT Press, 2003.

All of the articles in this issue mention one main cause of the multicore revolution that echoes, in a surprising way, an environmental concern very much in today's news – system overheating. The underlying physical system on which all software ecologies depend, i.e., the computer chip, has traditionally been designed in a way that required them to get hotter as they got faster, but now that process has reached its limit. The collision of standard processor designs with this thermal barrier, as well as with other stubborn physical limits, has forced processor architects to develop chip designs whose additional computing power can only be utilized by software that can effectively and efficiently exploit parallelism. Precious little of the software now in use has that capability. Precious few in the computing community have any idea of how to change the situation any time soon. As the hardware habitat for software adapted for serial processing declines, and the steep challenges of creating good parallel software become more and more evident, the consequences of the discontinuity thereby produced seem destined to reverberate through nearly every element of our software ecosystem, including libraries, algorithms, operating systems, programming languages, performance tools, programmer training, project organization, and so on.

Since the impacts of these changes are so broad, far reaching, and largely unknown, it is important in discussing them to have different points of view within the global software ecosystem represented. The sample of views presented here includes one from a group of academic researchers in high performance computing, and three from different industry niches. As one would expect, despite certain commonalities, each of them highlights somewhat different aspects of the situation.

The article from the more academic perspective, authored by Dennis Gannon, Geoffrey Fox, the late Ken Kennedy, and myself, focuses primarily on the relatively small but important habitat of Computational Science. Starting from the basic thesis that *science itself* now requires and has developed a software ecosystem that needs stewardship and investment, we provide a brief characterization of three main disruptors of the status quo: physical limits on clock rates and voltage, disparities between processor speed and memory bandwidth, and economic pressures encouraging heterogeneity at the high end. Since the HPC community has considerably more experience with parallel computing than most other communities, it is in a better position to communicate some lessons learned from science and engineering applications about scalable parallelism. The chief one is that *scalable parallel performance is not an accident*. We look at what these lessons suggest about the issues that commodity applications might face and draw out some of their future implications for the critical areas of numerical libraries and compiler technologies.

## Introduction

John Manferdelli of Microsoft explores the situation for commercial application and system programmers, where the relative paucity of experience with parallel computing is liable to make the "shock of the new," delivered by the multicore revolution, far more severe. After presenting David Patterson's compact formulation of how the performance of serial processing has been indefinitely barred from further improvements by the combined force of the "power wall," the "memory wall," and the "ILP wall," he describes several complementary approaches for getting more concurrency into programming practice. But to be successful in helping commercial developers across the concurrency divide, these new development tools and techniques will require improvements on other fronts, most especially in operating systems. He sketches an important preview of how we may expect operating systems to be adapted for concurrency, providing new approaches to resource sharing that allow different system subcomponents to have flexible access to dedicated resources for specialized purposes.

A view from the inside of the multicore revolution, offering an extended discussion of the critical factors of "balance" and "optimization" in processor design, is provided in the article by John McCalpin, Chuck Moore, and Phil Hester of Advanced Micro Devices (AMD). Their account of motivation for adopting a multicore design strategy is much more concrete and quantitative than the previous articles, as is only appropriate for authors who had to grapple with this historic problem at first hand and as a mission critical goal. They offer a fascinating peek inside the rationale for the movement to multicore, laying out the lines of reasoning and the critical tradeoffs and considerations (including things like market trends) that lead to different design points for processor manufacturers. Against this background, the speculations they offer about what we might expect in the near and mid-range future are bound to have more credibility than similar exercises by others not so well positioned.

Finally, the article by David Turek of IBM highlights a distinctly different but equally important strand in the multicore revolution: the introduction of new hybrid multicore architectures and their application to supercomputing architectures. A conspicuous example of this significant trend is use of the Cell Broadband Engine processor, created by an industry consortium to power Sony's next generation PlayStations, in the construction of novel supercomputer designs, like the Roadrunner system at Los Alamos National Laboratory. Such hybrid systems provide convincing illustrations of how unexpected combinations of technological and economic forces in the software ecosystem can combine to produce new innovation. The dark side of this trend toward heterogeneity, however, is that it severely complicates the planning process of small ISV's, who have only scarce resources to apply to the latest hardware innovations.

It is hard to read the articles in this issue of *CTWatch Quarterly* without being driven to the conclusion, agreed upon by many other leaders in the field, that modern software ecosystems are about to be destabilized, not to say convulsed, by a major transformation in their hardware substrate. Over time, this may actually improve the health of the software by changing people's attitudes about the value of software. There have long been complaints, especially in the HPC community, that software is substantially undervalued, with inadequate investments, beyond the initial research phase, in software hardening, enhancement, and long term maintenance. New federal programs, such as the NSF's Software Development for Cyberinfrastructure (SDCI), represent a good first step in recognizing that good software has become absolutely essential to productivity in many areas. Yet I believe the multicore revolution, which is now upon us, will drive home the need to make that recognition into a guiding principle of our national research strategy. For that reason alone, but for many others as well, the CTWatch community, and the scientific computing community in general, will miss the incandescent presence of Ken Kennedy, who died earlier this month. The historic challenges we are about to confront will require the very kind of visionary leadership for which Ken had all the right stuff, as he showed over and over again during his distinguished career. I will miss my friend.

# The Impact of Multicore on Computational Science Software

Jack Dongarra
University of Tennessee
Oak Ridge National Laboratory
Dennis Gannon
Indiana University
Geoffrey Fox
Indiana University
Ken Kennedy
Rice University

## 1. Introduction

The idea that computational modeling and simulation represents a new branch of scientific methodology, alongside theory and experimentation, was introduced about two decades ago. It has since come to symbolize the enthusiasm and sense of importance that people in our community feel for the work they are doing. But when we try to assess how much progress we have made and where things stand along the developmental path for this new "third pillar of science," recalling some history about the development of the other pillars can help keep things in perspective. For example, we can trace the systematic use of experiments back to Galileo in the early 17th century. Yet for all the incredible successes it enjoyed over its first three centuries, the experimental method arguably did not fully mature until the elements of good experimental design and practice were finally analyzed and described in detail by R. A. Fisher and others in the first half of the 20th century. In that light, it seems clear that while Computational Science has had many remarkable youthful successes, it is still at a very early stage in its growth.

Many of us today who want to hasten that growth believe that the most progressive steps in that direction require much more community focus on the vital core of Computational Science: s*oftware and the mathematical models and algorithms it encodes*. Of course the general and widespread obsession with hardware is understandable, especially given exponential increases in processor performance, the constant evolution of processor architectures and supercomputer designs, and the natural fascination that people have for big, fast machines. But when it comes to advancing the cause of computational modeling and simulation as a new part of the scientific method, there is no doubt that the complex software "ecosystem" it requires must take its place on the center stage.

At the application level, the science has to be captured in mathematical models, which in turn are expressed algorithmically and ultimately encoded as software. Accordingly, on typical projects the majority of the funding goes to support this translation process that starts with scientific ideas and ends with executable software, and which over its course requires intimate collaboration among domain scientists, computer scientists and applied mathematicians. This process also relies on a large infrastructure of mathematical libraries, protocols and system software that has taken years to build up and that must be maintained, ported, and enhanced for many years to come if the value of the application codes that depend on it are to be preserved and extended. The software that encapsulates all this time, energy, and thought routinely outlasts (usually by years, sometimes by decades) the hardware it was originally designed to run on, as well as the individuals who designed and developed it.

Thus the life of Computational Science revolves around a multifaceted software ecosystem. But today there is (and should be) a real concern that this ecosystem of Computational Science, with all its complexities, is not ready for the major challenges that will soon confront the field. Domain scientists now want to create much larger, multi-dimensional applications in which a variety of previously independent models are coupled together, or even fully integrated. They hope to be able to run these applications on Petascale systems with tens of thousands of processors, to extract all the performance these platforms can deliver, to recover automatically from the processor failures that regularly occur at this scale, and to do all this without sacrificing good programmability. This vision of what Computational Science wants to become contains numerous unsolved and exciting problems for the software research community. Unfortunately, it also highlights aspects of the current software environment that are either immature or under funded or both.[1]

[1] Post, D. E., Votta, L. G. "Computational Science Demands a New Paradigm," *Physics Today*, vol. 58, no. 1, pp. 35-41, January, 2005.

## 2. The Challenges of Multicore

It is difficult to overestimate the magnitude of the discontinuity that the high performance computing (HPC) community is about to experience because of the emergence of the next generation of multi-core and heterogeneous processor designs.[2] For at least two decades, HPC programmers have taken it for granted that each successive generation of microprocessors would, either immediately or after minor adjustments, make their old software run substantially faster. But three main factors are converging to bring this "free ride" to an end. First, system builders have encountered intractable physical barriers – too much heat, too much power consumption, and too much leaking voltage – to further increases in clock speeds. Second, physical limits on the number of pins and bandwidth on a single chip mean that the gap between processor performance and memory performance, which was already bad, will get increasingly worse. Finally, the design trade-offs being made to address the previous two factors will render commodity processors, absent any further augmentation, inadequate for the purposes of tera- and petascale systems for advanced applications. This daunting combination of obstacles has forced the designers of new multi-core and hybrid systems, searching for more computing power, to explore architectures that software built on the old model are unable to effectively exploit without radical modification.

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

But despite the rapidly approaching obsolescence of familiar programming paradigms, there is currently no well understood alternative in whose viability the community can be confident. The essence of the problem is the dramatic increase in complexity that software developers will have to confront. Dual-core machines are already common, and the number of cores is expected to roughly double with each processor generation. But contrary to the assumptions of the old model, programmers will not be able to consider these cores independently (i.e., multi-core is *not* "the new SMP") because they share on-chip resources in ways that separate processors do not. This situation is made even more complicated by the other non-standard components that future architectures are expected to deploy, including mixing different types of cores, hardware accelerators, and memory systems. Finally, the proliferation of widely divergent design ideas illustrates that the question of how to best combine all these new resources and components is largely unsettled. When combined, these changes produce a picture of a future in which programmers must overcome software design problems that are vastly more complex and challenging than in the past in order to take advantage of the much higher degrees of concurrency and greater computing power that new architectures will offer.

### 2.1 Main factors driving the multi-core discontinuity

Among the various factors that are driving the momentous changes now occurring in the design of microprocessors and high end systems, three stand out as especially notable: 1) *the number of transistors on the chip will continue to double roughly every 18 months, but the speed of processor clocks will not continue to increase*; 2) *the number of pins and bandwidth on CPUs are reaching their limits*; and 3) *there will be a strong drift toward hybrid systems for petascale (and larger) systems*. The first two involve fundamental physical limitations that nothing currently on the horizon is likely to overcome. The third is a consequence of the first two, combined with the economic necessity of using many thousands of CPUs to scale up to petascale and larger systems. Each of these factors has a somewhat different effect on the design space for future programming:

1) *More transistors and slower clocks mean multi-core designs and more parallelism required* – The *modus operandi* of traditional processor design – increase the transistor density, speed up the clock rate, raise the voltage – has now been blocked by a stubborn set of physical barriers – too much heat produced, too much power consumed, too much voltage leaked. Multi-core designs are a natural response to this situation. By putting multiple processor cores on a single

die, architects can continue to increase the number of gates on the chip without increasing the power densities. But since excess heat production means that frequencies cannot be further increased, deep-and-narrow pipeline models will tend to recede as shallow-and-wide pipeline designs become the norm. Moreover, despite obvious similarities, multi-core processors are not equivalent to multiple-CPUs or to SMPs. Multiple cores on the same chip can share various caches (including TLB!) and they certainly share the bus. Extracting performance from this configuration of resources means that programmers must exploit increased thread-level parallelism (TLP) and efficient mechanisms for inter-processor communication and synchronization to manage resources effectively. The complexity of parallel processing will no longer be hidden in hardware by a combination of increased instruction level parallelism (ILP) and deep-and-narrow pipeline techniques, as it was with superscalar designs. It will have to be addressed in software.

2) *Thicker "memory wall" means that communication efficiency will be even more essential* – The pins that connect the processor to main memory have become a strangle point, with both the rate of pin growth and the bandwidth per pin slowing down, if not flattening out. Thus the processor- to-memory performance gap, which is already approaching a thousand cycles, is expected to grow, by 50% per year according to some estimates. At the same time, the number of cores on a single chip is expected to continue to double every 18 months, and since limitations on space will keep the cache resources from growing as quickly, the cache per core ratio will continue to go down. Problems of memory bandwidth, memory latency, and cache fragmentation will, therefore, tend to get worse.

3) *Limitations of commodity processors will further increase heterogeneity and system complexity* - Experience has shown that tera- and petascale systems must, for the sake of economic viability, use commodity off-the-shelf (COTS) processors as their foundation. Unfortunately, the trade-offs that are being (and will continue to be) made in the architecture of these general purpose multi-core processors are unlikely to deliver the capabilities that leading edge research applications require, even if the software is suitably modified. Consequently, in addition to all the different kinds of multithreading that multi-core systems may utilize – at the core-level, socket-level, board-level, and distributed memory level – they are also likely to incorporate some constellation of special purpose processing elements. Examples include hardware accelerators, GPUs, off-load engines (TOEs), FPGAs, and communication processors (NIC-processing, RDMA). Since the competing designs (and design lines) that vendors are offering are already diverging, and mixed hardware configurations (e.g., Los Alamos Roadrunner, Cray BlackWidow) are already appearing, the hope of finding a common target architecture around which to develop future programming models seems at this point to be largely forlorn.

We believe that these major trends will define, in large part at least, the design space for scientific software in the coming decade. But while it may be important for planning purposes to describe them in the abstract, to appreciate what they mean in practice, and therefore what their strategic significance may be for the development of new programming models, one has to look at how their effects play out in concrete cases. Below we describe our early experience with these new architectures, both how they render traditional ideas obsolete, and how innovative techniques can exploit their parallelism and heterogeneity to address these problems.

### 2.2 Lessons from Science and Engineering applications

We need to understand how to broaden the success in parallelizing science and engineering to the much larger range of applications that could or should exploit multicore chips. In fact, this broader set of applications must get good speedups on multicore chips if Moore's law is to continue while we move from the single CPU architecture and clock speed improvements that drove past exponential

performance increases to performance improvement driven by increasing cores per chip. We will focus on "scalable" parallelism where a given application can get good performance on 16-32 cores or more. On general principles backed up by experience from scientific and engineering, lessons from a small number of cores are only a good harbinger for scaling to larger systems if they are backed up with a good model of the parallel execution. So in this section, we analyze lessons from science and engineering on scalable parallelism – how one can get speed-up that is essentially proportional to the number of cores as one scales from 4-16-128 and more cores. These lessons include, in particular, a methodology for measuring speedup and identifying and measuring bottlenecks, which is intrinsically important and should be examined and extended for general multicore applications. Note that multicore offers not just an implementation platform for science and engineering but an opportunity for improved software environments sustained by a broad application base. Thus there are clear mutual benefits in incorporating lessons and technologies developed from scalable parallel science and engineering applications into broader commodity software environments.

Let us discuss the linear algebra example described below in this article. Linear algebra illustrates the key features of most scalable, parallel science and engineering applications. These applications have a "space" with degrees of freedom (for linear algebra, the space is formed by vectors and matrices and the matrix/vector elements are degrees of freedom). This space is divided into parts, and each core of the processor is responsible for "evolving" its part. The evolution consists of multiple algorithmic steps, time-steps or iterations. The parts synchronize with each other at the end of each step, and this synchronization leads to "overhead." However, with carefully designed parallel algorithms, this overhead can be quite small with the speedup $S$ for an application on $N$ cores having the form $\varepsilon N$, where the efficiency often has the form $(1- c/n^{\alpha})$, where $n$ is the (grain) size of the part in each core, and $c$ and $\alpha$ are application and hardware dependent constants. This form has the well-understood scaled speedup consequence that the speedup is directly proportional to the number of cores $N$ as long as the grain size $n$ is constant; this implies that the total problem size $nN$ also grows proportionally to $N$. The exponent $\alpha$ is 0.5 for linear algebra so the efficiency will decrease as $N$ increases for a fixed total problem size when $n$ is proportional to $1/N$. The success of parallel computing in science and engineering partly reflects that users do need to run larger and larger problems, so grain sizes are not decreasing as we scale up machines – in fact as both the number of nodes and memory sizes per CPU have grown in successive generations of parallel computers, the grain sizes are increasing not decreasing. It is important to ask if commodity applications have this characteristic; will they scale up in size as chips increase in number of cores? This class of applications consists of successive phases; each compute phase is followed by a communication (synchronization) phase and this pattern repeats. Note, synchronization is not typically a global operation but rather achieved by each part (core) communicating with other parts to which it is linked by the algorithm.

We take a second example, which is deliberately chosen to be very different in structure. Consider a typical Web server farm where parallelism occurs from multiple users accessing some sort of resource. This is typical of job scheduling on multiple computers or cores. Here, efficient operation is not achieved by linked, balanced processes but rather by statistical balancing of heterogeneous processes assigned dynamically to cores. These applications are termed "pleasingly" or "embarrassingly" parallel and quite common; as the jobs are largely independent, one does not require significant communication, and Grids are a popular implementation. Such (essentially) pleasingly parallel applications include data analysis of the events from the Large Hadron Collider (LHC) but also, for example, genetic algorithms for optimization.

There are of course other types of scalably parallel applications, but the two described illustrate an important characteristic – namely that good parallel performance is not an "accident." Rather, it is straightforward (albeit often hard!) to build a model for applications of these classes to predict performance, i.e., for the first application class to predict the constants $c$ and $\alpha$ and hence what grain size $n$ is needed to meet speedup goals. Science and engineering applications involve starting

with an abstract description of the problem (say the Navier-Stokes partial differential equations for computational fluid dynamics) and mapping this into a numerical formulation involving multiple steps such as a discretization and the choice of a solver for the sparse matrix equations. It is usually easiest to model the performance at this stage, as the description is manifestly parallelizable. Then one chooses a software model (and implicitly often at run time) to express the problem, and here one chooses between technologies like MPI and openMP or compiler discerned parallelism. Note, there is no known universal way of expressing a clearly parallel model in software that is guaranteed to preserve the natural parallelism of the model. Formally, the mapping is not invertible and the parallelism may or may not be discovered statically. Further, it may or may not be realistic even to discover the parallelism dynamically as the application runs. The differences between abstract description, numerical model, and software and hardware platforms are significant and lead to some of the hardest and most important problems in parallel computing. Explicit messaging passing with MPI is successful even though tough for the programmer. It expresses the inherent parallelism; many languages originating with CMFortran and HPF and continued with Global array approaches, and the Darpa HPCS languages permit the expression of collective operations and data decompositions that allow the parallelism of many but not all problems. OpenMP provides hints that help compilers fill in information lost in the translation of numerical models to software. The success of fusion and tiling strategies described below for efficient compilation is a consequence of the characteristics, such as geometric locality, typical of problems whose parallelism is "not accidental." The development of powerful parallel libraries and dynamic run time strategies such as inspector-executor loops allows run time to help efficient parallel execution. The science and engineering community is still debating the pluses and minuses of these different approaches, but the issues are clear and the process of identifying and expressing parallelism understood.

It seems important to examine the process for important commodity applications. Can one identify the same process which we note has not typically been to start with a lump of software with unclear parallel properties? Some applications of interest such as machine learning or image processing are related to well understood, parallel applications, and we can be certain that parallelism is possible with good efficiency if the problem is large enough. Are we aiming to parallelize applications with clear models that suggest good performance or are we hoping for a miracle by tossing a bunch of interacting threads on a bunch of cores? The latter could in fact be very important but likely to require a different approach and different expectations from the cases where parallelism is "no accident." Applications based on large numbers of concurrent threads, all running in the same address space, are not uncommon in commercial software such as portals and web services, games, and desktop application frameworks. In these applications, data structures are shared and communication is through explicit synchronization involving locks and semaphores. Unfortunately, locked, critical sections do not scale well to large numbers of cores, and alternative solutions are required. One extremely promising approach is based on a concept called software transactional memory (STM),[3] where critical sections are not locked, but conflicts are detected and bad transactions are rolled back. While some argue that for STM to work efficiently it will require additional hardware support, the jury is still out. It may be that smart compilers and the runtime architecture will address efficiency issues without changes to the hardware. Note, even if the same approaches can be used for commodity and science and engineering, we can well find a different solution as we will now find a broad commodity base rather than a niche application field for parallel applications; this could motivate the development of powerful software development environments or HPF style languages that were previously not economic.

[3] Herlihy, M., Luchangco, V., Moir, M., Scherer III, W. N. "Software Transactional Memory for Dynamic-Sized Data Structures," *Proceedings of the Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (PODC), 92–101. July 2003.

### 2.3 Free ride is over for HPC software: Case of LAPACK/ScaLAPACK

One good way to appreciate the impact and significance of the multi-core revolution is to examine its effect on software packages that are widely familiar. The LAPACK/ScaLAPACK libraries for linear algebra fit that description. These libraries, which embody much of our work in the adaptation of block partitioned algorithms to parallel linear algebra software design, have served the HPC and Computational Science community remarkably well for 20 years. Both LAPACK and ScaLAPACK apply the idea of blocking in a consistent way to a wide range of algorithms in linear algebra (LA), including linear systems, least square problems, singular value decomposition, eigenvalue decomposition, etc., for problems with dense and banded coefficient matrices. ScaLAPACK also addresses the much harder problem of implementing these routines on top of distributed memory architectures. Yet it manages to keep close correspondence to LAPACK in the way the code is structured or organized. The design of these packages has had a major impact on how mathematical software has been written and used successfully during that time. Yet, when you look at how these foundational libraries can be expected to fair on large-scale multi-core systems, it becomes clear that we are on the verge of a transformation in software design, at least as potent as the change engendered a decade ago by message passing architectures, when the community had to rethink and rewrite many of its algorithms, libraries, and applications.

Historically, LA methods have put a strong emphasis on *weak scaling* or *isoscaling* of algorithms, where speed is achieved when the number of processors are increased while the problem size per processor is kept constant, effectively increasing the overall problem size.[4] This measure tells us when we can exploit parallelism to solve larger problems. In this approach, increasing the speed of a single processing element should decrease the time to solution. But in the emerging era of multiprocessors, although the number of processing elements (i.e., cores) in systems will grow rapidly (exponentially, at least for a few generations), the computational power of individual processing units is likely to be reduced. Since the speed of individual processors will decline, we should expect that problems reaching their scaling limits on a certain number of processors will require *increased time to solution on the next generation of architectures*. In order to address the problem, emphasis has to be shifted from weak to *strong scaling*, where speed is achieved when the number of processors increased while *the overall problem size is kept constant*, which effectively decreases the problem size per processor. But to achieve this goal we have to examine methods to exploit parallelization at much finer granularity than traditional approaches employ.

[4] Dongarra, J. J., Luszczek, P., Petitet, A. "The LINPACK Benchmark: Past, Present, and Future," *Concurrency and Computation: Practice and Experience* vol. 15, no. 9, pp. 803-820, August, 2003. http://www.netlib.org/benchmark/hpl/

The standard approach to parallelization of numerical linear algebra algorithms for both shared and distributed memory systems, utilized by the LAPACK/ScaLAPACK libraries, is to rely on a parallel implementation of BLAS - threaded BLAS for shared memory systems and PBLAS for distributed memory systems. Historically, this approach made the job of writing hundreds of routines in a consistent and accessible manner doable. But although this approach solves numerous complexity problems, it also enforces a very rigid and inflexible software structure, where, *at the level of LA, the algorithms are expressed in a serial way*. This obviously inhibits the opportunity to exploit inherently parallel algorithms at finer granularity. This is shown by the fact that the traditional method is successful mainly in extracting parallelism from Level 3 BLAS; in the case of most of the Level 1 and 2 BLAS, however, it usually fails to achieve speedups and often results in slowdowns. It relies on the fact that, for large enough problems, the $O(n^3)$ cost of Level 3 BLAS dominates the computation and renders the remaining operations negligible. The problem with encapsulating parallelization in the BLAS/PBLAS in this way is that it requires a heavy synchronization model on a shared memory system and a heavily synchronous and blocking form of collective communication on distributed memory systems with message passing. This paradigm will break down on the next generation architectures, because it relies on coarse grained parallelization and emphasizes weak scaling, rather than *strong scaling*.

## 2.4 Compiling for Multicore

Although many applications can make significant performance gains through the use of tuned library kernels, such as those from dense linear algebra, there will be application components needing a more general strategy for mapping to multicore. In such components, it is important to avoid explicit tuning to a particular number of cores or hierarchy of cache series. Performance of these mappings, while hiding underlying details of the target computing platform, has been a main goal of research in compiler technology for the past two decades. Much of that work has been guided by the lessons from the tuning of library kernels. The compiler attempts to generalize tuning strategies for library kernels, applying them to arbitrary nests of loops.

Multicore chips amplify the need for careful mapping because the caches are shared at one or more levels on the chip, but bandwidth on and off the chip is limited and unlikely to scale with the number of cores. This reuse of data in on-chip cache and sharing data in cache between multiple cores are key performance improving strategies. Two main compiler transformations studied since the late 1980s are key to successfully exploring these opportunities:

1. *Tiling* is the practice of breaking long loop nests into blocks of computation that fit neatly into a single level of cache. This transformation, applied at several levels of the hierarchy, has been a cornerstone of optimization for dense linear algebra kernels for over two decades. The trick is to apply it with equal success to general loop nests.
2. *Fusion* of loop nests is a strategy that takes different loops that use the same arrays and brings them together into a single loop nest to enhance cache reuse. Of course, this has a significant interaction with tiling, often forcing the use of smaller tiles.

These transformations have been broadly studied and are used in most commercial compilers today. However, the interactions between them are tricky to manage because of another feature of modern caches: data blocks are not simply stored anywhere in cache; instead, each block is mapped to an "associativity group" that is typically small in size, say two to eight blocks. Thus it is possible, in a nearly empty cache, that the hardware would evict a block that is still needed because there are too many other in-use blocks that map to the same group. The next time the evicted block is needed, it suffers a costly "miss" that requires fetching the block from memory. Such an event is called a *conflict miss* to distinguish it from a *capacity miss*, which occurs when the cache is full.

Reduction of conflict misses is very tricky because of its interactions with tiling and fusion – too much fusion and tiles that are too large, or even tiles of the wrong dimension can increase conflict misses.

Dealing with difficult trade-offs like these has become so complex that a significant amount of research is being invested in automatic tuning of loop optimizations. In this approach, the compiler runs loop nests on small data sets in advance to determine the best tile sizes and fusion configurations. Though expensive in computer time, it is far better than hand tuning to each new multicore chip. It is worth noting that the first autotuning work was applied to kernels from dense and sparse linear algebra,[5] along with libraries for the Fast Fourier Transform (FFT). Now the work is being expanded to handle other libraries and whole applications.

Finally, we observe that shared caches suggest that it may be preferable to pipeline computations on multicore chips. In this approach, two loops that might otherwise be fused are run on two different cores with synchronization and staging through the shared cache. This makes it possible to retain the benefits of fusion without shrinking tile sizes, at a cost of synchronization and pipeline start up delay.

[5] Kurzak, J., Dongarra, J. J. "Pipelined Shared Memory Implementation of Linear Algebra Routines with Lookahead - LU, Cholesky, QR." In *Workshop on State-of-the-Art in Scientific and Parallel Computing*, Umea, Sweden, June, 2006.

This final observation suggests that new programming models based on functional composition may be a great way to develop applications for multicore chips today and in the future.

## 3. The Future

Advancing to the next stage of growth for computational simulation and modeling will require us to solve basic research problems in Computer Science and Applied Mathematics at the same time as we create and promulgate a new paradigm for the development of scientific software. To make progress on both fronts simultaneously will require a level of sustained, interdisciplinary collaboration among the core research communities that, in the past, has only been achieved by forming and supporting research centers dedicated to such a common purpose. We believe that the time has come for the leaders of the Computational Science movement to focus their energies on creating such software research centers to carry out this indispensable part of the mission.

# The Many-Core Inflection Point for Mass Market Computer Systems

John L. Manferdelli
Microsoft Corporation

## Preface

Major changes in the commercial computer software industry are often caused by significant shifts in hardware technology. These changes are often foreshadowed by hardware and software technology originating from high performance, scientific computing research. Innovation and advancement in both communities have been fueled by the relentless, exponential improvement in the capability of computer hardware over the last 40 years and much of that improvement (keenly observed by Gordon Moore and widely known as "Moore's Law") was the ability to double the number of microelectronic devices that could be crammed onto a constant area of silicon (at a nearly constant cost) every two years or so. Further, virtually every analytical technique from the scientific community (operations research, data mining, machine learning, compression and encoding, signal analysis, imaging, mapping, simulation of complex physical and biological systems, cryptography) has become widely deployed, broadly benefiting education, health care and entertainment as well as enabling the world-wide delivery of cheap, effective and profitable services from eBay to Google.

In stark contrast to the scientific community, commercial application software programmers have not, until recently, had to grapple with massively concurrent computer hardware. While Moore's law continues to be a reliable predictor of the aggregate computing power that will be available to commercial software, we can expect very little improvement in serial performance of general purpose CPUs. So if we are to continue to enjoy improvements in software capability at the rate we have become accustomed to, we must use parallel computing. This will have a profound effect on commercial software development including the languages, compilers, operating systems, and software development tools, which will in turn have an equally profound effect on computer and computational scientists.

## Computer Architecture: What happened?

Power dissipation in clocked digital devices is proportional to the clock frequency, imposing a natural limit on clock rates. While compensating scaling has enabled commercial CPUs to increase clock speed by a factor of 4,000 in the last 10 years, the ability of manufacturers to dissipate heat has reached a physical limit. Leakage power dissipation gets worse as gates get smaller, because gate dielectric thicknesses must proportionately decrease. As a result, a significant increase in clock speed without heroic (and expensive) cooling is not possible. Chips would simply melt. This is the "Power Wall" confronting serial performance, and our back is firmly against it: Significant clock-frequency increases will not come without heroic measures, or materials technology breakthroughs.

Not only does clock speed appear to be limited, but memory performance improvement increasingly lags behind processor performance improvement. This introduces a problematic and growing memory latency barrier to computer performance improvements. To try to improve the "average memory reference" time to fetch or write instructions or data, current architectures have ever growing caches. Cache misses are expensive, causing delays of hundreds of (CPU) clock cycles. The mismatch in memory speed presents a "Memory Wall" for increased serial performance.

In addition to the performance improvements that have arisen from frequency scaling, hardware engineers have also improved performance, on average, by having duplicate hardware speculatively execute future instructions before the results of current instructions are known, while providing hardware safeguards to prevent the errors that might be caused by out of order execution.[1]

[1] A related ILP technique breaks instructions into multiple cycles and attempts to execute different parts of successive instructions simultaneously, branch prediction is also needed here.

Unfortunately, branches must be "guessed" to decide what instructions to execute simultaneously (if you guess wrong, you throw away this part of the result) and data dependencies may prevent successive instructions from executing in parallel, even if there are no branches. This is called Instruction Level Parallelism (ILP). A big benefit of ILP is that existing programs enjoy performance benefits without any modification. But ILP improvements are difficult to forecast since the "speculation" success is difficult to predict, and ILP causes a super-linear increase in execution unit complexity (and associated power consumption) without linear speedup. Serial performance acceleration using ILP has also stalled because of these effects.[2] This is the "ILP Wall."

[2] ILP also increases the "energy per useful computation" because of the discarded results and much larger controllers.

David Patterson of Berkeley has a formulaic summary of the serial performance problem: "The power wall + the memory wall + the ILP wall = a brick wall for serial performance." Thus, the heroic line of development followed by materials scientists and computer designers to increase serial performance now yields diminishing returns. Computer architects have been forced to turn to parallel architectures to continue to make progress. Parallelism can be exploited by adding more independent CPUs, data-parallel execution units, additional registers sets (hardware threads), more independent memory controllers to increase memory bandwidth (this requires more output pins) and bigger caches. Computer architects can also consider incorporating different execution units, which dramatically improve some computations but not others (e.g., GPU like units that excel at structured data parallelism and streaming execution units with local memory as Cray did in many of its early machines).[3] Heterogeneity need not only mean completely different "abstract" execution unit models but may include incorporating computation engines with the same instruction set architecture, but different performance and power consumption characteristics. All of these take advantage of dramatically higher on-chip interconnect data rates.

[3] The cell processor does this.

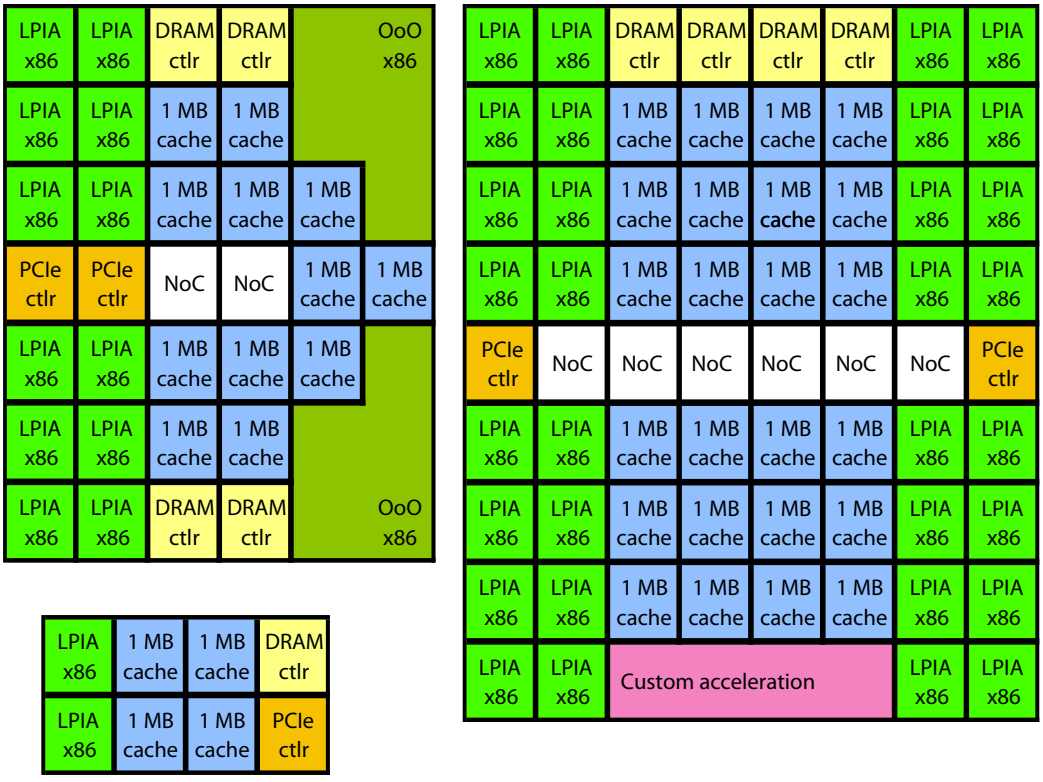


Figure 1. (a, b, c – clockwise from upper left). Sample client, server, embedded chips.

Moore's law will grant computer architects ever more gates for the foreseeable future, and the challenge is to use them to deliver performance and power characteristics fit for their intended purpose. Figure 1 below illustrates a few hardware design choices. In 1 (a), a client configuration might consist of two large "out of order" cores (OoC) incorporating all the ILP of current processors to run existing programs together with many smaller "in-order" cores (IoC) for programs that can take advantage of highly parallel software. Why many IoCs rather than correspondingly fewer of the larger OoCs? The reason is that spending gates on out-of order has poorer performance returns than simpler in-order cores, *if* parallel software can scale with cores. The server configuration in 1 (b) incorporates many more IoCs and a "custom" core (say a crypto processor). Finally, multi-core computers are not just beneficial for raw performance but also reliability and power management so embedded processors will also undergo an architecture shift as illustrated in 1(c).

## Developing Software

While the foregoing hardware architecture offers much more computing power, it makes writing software that can fully benefit from the hardware potentially much harder.

In scientific applications, improved performance has historically been achieved by having highly trained specialists modify existing programs to run efficiently as new hardware was provided.[4] In fact, even re-writing existing programs in this environment was far too costly, and most organizations focused the specialists on rewriting small portions of the "mission critical" programs, called kernels. In the "good case," the mission critical applications spent 80 or 90% of their time in these kernels and the kernels represented a few percent of the application code. Thus making a kernel ten times faster could mean a nearly ten-fold performance improvement. Even so, this rewriting was time consuming, and organizations had to balance the risk of introducing subtle bugs into well tested programs against the benefit of increased speed at every significant hardware upgrade. All bets were off if the organization did not have the source code for the critical components.

[4] In fairness, compilers, runtimes, frameworks and libraries were also improved to try to ameliorate this problem. See for example, Allen and Kennedy in the references.

By contrast, commercial vendors, thanks to the chip manufacturers who managed to rapidly improve the serial performance while maintaining the same hardware instruction set architecture, have been habituated to a world where all existing programs get faster with each new hardware generation. Further, software developers could confidently build new innovative software, which barely ran on then current hardware, knowing that it would run quite well on the next generation machine at the same cost. This will no longer occur for serial codes, but the goal of new software development tools must be to retain this very desirable characteristic as we move into the era of many-core computing. If we are successful, then building your software with these new tools and then faster hardware (or even just adding more hardware) will improve performance without further application programmer intervention.

In order to benefit from rapidly improving computer performance (and we all want that) and to retain the "write one, run faster on new hardware" paradigm, commercial software and scientific software must change their software development and system support.[5] To achieve this, software development systems and supporting software must enable a significant portion of the programming community to construct parallel applications. There are several complementary approaches that may help us achieve this.

[5] Larus, J., Sutter, H. "Software and the Concurrency Revolution," *ACM Queue*, Vol. 3, No. 7, pp 54–62, September 2005.

1. **Encapsulate domain specific knowledge in reusable parallel component.** The most effective way to deploy concurrency without needing to disturb the programming model for most developers is to encapsulate concurrency with domain knowledge of common reusable library components. This approach mirrors the use of numerical kernels beloved by computational scientists, but moves them into the world of general-purpose computing. This technique is

ideal when it works, although composing such libraries requires better synchronization and resource-management techniques.

2. **Integrate concurrency and coordination into traditional languages.** Current languages have little or no support for expressing or controlling parallelism. Instead, programmers must use libraries or OS facilities. Other language features, like the use of for/while loops and linked lists, obscure potential parallelism from the compiler. To build parallel applications, we will need to extend traditional sequential languages with new features to allow programmers to explicitly guide program decomposition into parallel subtasks, as well as provide atomicity and isolation as those subtasks interact with shared data structures. Transactional memory[6] shows promise here and also provides a way towards composing independently developed software components.
3. **Raising semantic level to eliminate explicit sequencing.** For many developers, we want to avoid the need to use procedural languages and use domain-specific systems based on rules or constraints. More declarative styles specify intent rather than sequencing of primitives and thus inherently permit parallel implementations that leverage the concurrency and transaction mechanisms of the system. SQL is a common example of this: it is declarative, and correctly written SQL can be executed much faster, and without modification, when supporting software (query optimizers) that adapts to different, more parallel hardware.

[6] Larus, J. R. , Rajwar , R. *Transactional Memory.* Morgan & Claypool, 2006.

However, to fully exploit parallelism, programmers must understand a parallel execution model, develop parallel algorithms, and be equipped with much better tools to develop, test and automatically tune performance. This requires education as well as software innovation. Compilers, which bridge between intent-oriented features and the underlying execution model of the system, must incorporate idioms to explicitly identify parallel tasks as well as optimization techniques to identify and schedule implicitly parallel tasks discovered by it.[7] Program analysis and testing are hard enough for sequential programs and are much harder in parallel programming. We must find mechanisms that contain concurrency and isolate threads and use those to make testing more robust. We have seen dramatic improvements in static analysis tools that identify software defects, reduce test burden and improve reliability. These techniques are being extended to incorporate identification of concurrency problems. Debuggers must evolve from the low-level machine model back to a more common and familiar model that a developer can reason about correctly and effectively. Finally, the need for tools for performance analysis to help identify bottlenecks will become crucial as we face the possibility of two orders of magnitude difference between optimized and naïve algorithms.

[7] Allen, R., Kennedy, K. *Optimizing Compilers for Modern Architectures.* Elsevier, 2002.

## System Software Architecture

Many-core computers are more like "data-centers-on-a-chip" than traditional computers. System software will change to effectively manage resources on these systems while decomposing and rationalizing the system software function to provide more reliability and manageability. General purpose computer operating systems (which have not fundamentally changed since system and application software separated with the advent of "time shared" computers in the 1950's) will change as much as development tools.

To understand why, consider the following. Supercomputing applications are typically assigned dedicated system wide resources for each application run. This allows applications to tune algorithms to available resources: knowledge of the actual CPU resources available to the application at runtime, as well as memory, can drastically improve a sophisticated application's performance (database systems do a good job of this right now and too so often avoid, or out and out deceive, current operating systems to control real resources). By contrast, most commercial operating systems "time multiplex" the hardware resources[8] to provide good utilization of expensive resources and anticipate that an application will run on a fairly narrow spectrum of architectures. Older oper-

[8] This made perfect sense when CPU's were expensive and memory subsystems were roughly comparable in speed.

ating systems also suffer from service, program and device isolation models, which are no longer appropriate but made perfect sense given earlier assumptions:

1. Current operating systems manage devices with a uniform device driver model and, if all such drivers are in the same address space, this simplifies I/O programming for applications and optimizes performance but creates huge OS kernels with management and security problems.
2. Time shared operating systems model security under a single authority (the "root" or "Administrator") who personally installs all software that is shared or requires OS modification software, knows all the users personally, and can determine a uniform security and resource allocation policy across (relatively simple) user programs. Today's computers operate in multiple trust domains, and different programs need different levels of protection and security policy; there are so many devices and some are so complex that no single authority can possibly uniformly and safely manage them. Right now, a buggy device driver used by one program jeopardizes all programs, while highly performant applications using special hardware (high speed graphics, for example) prefer to manage the device directly without incurring the sometimes catastrophic degradation incurred by "context switches" in the OS.
3. Homogenous operating systems are usually designed for one of three modes or operation: high throughput, high reliability or high real-time guarantees. General purpose OSs fall into the first category, an OS designed to run a central phone switch in a major location falls into the second, and an entertainment or media device falls into the third. It is difficult to design a single scheduler that serves all three environments, but these computers will have each of these applications running simultaneously.
4. Most general purpose operating system configurations contain "everything any application could want." This has dramatically increased OS complexity by decreasing utility and slowing down all application development.
5. Most operating systems, again to simplify programming, have a "chore scheduling" model in which each independent "thread of execution" is scheduled by the OS. This means that every "chore" switch incurs a context switch into the kernel, which is very expensive. The OS scheduler, which knows nothing about the individual application, must "guess" as to what's best to do next. Historically, operating systems have given their applications one millisecond to run before interrupting, and rescheduling and switching to another thread might have taken 100 instructions. On a 1 MIP machine, this means a thread can run about 1000 instructions of useful work so system overhead was a very acceptable 10%. On a very fast machine, a millisecond accounts for a few million of instructions and it is very hard to write general purpose programs where this "quantum" of instructions yields a highly concurrent duty cycle. This forces programs with high concurrency to structure themselves into bigger, less parallel subtasks or suffer catastrophic performance. The solution is to let a "runtime," linked into the application, handle the vast majority of "chore switches" without OS intervention. These runtimes can have very detailed knowledge of the actual hardware configuration and make resource and scheduling decisions appropriately.

Many core operating systems will incorporate a hypervisor, a small and very reliable component that hosts many different operating systems (or copies of the same OS with different performance or security characteristics). Hypervisors perform the relatively slowly change in "space sharing" of resources. For example, a hypervisor might simultaneously dedicate a core for long periods of time to a multimedia OS partition/application combination and assign I/O devices to it, host an older, buggier version of an OS for compatibility, host a tightly controlled corporate partition, host a game partition requiring strong performance guarantees, and host a loosely controlled partition for web browsing on the same hardware. Each partition can be sure of both performance and security isolation, and one partition need not incur the performance, fragility or security characteristics of another.

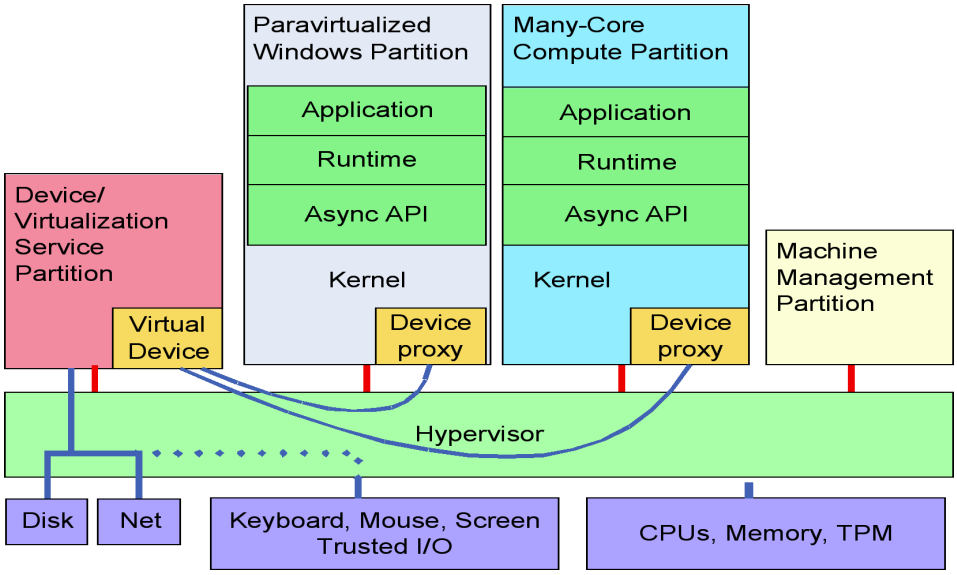


Figure 2. Many Core System Software Architecture.

A many-core system stack (hypervisor, OS kernel, user mode run-time) *must effectively assign resources securely and host concurrent operating environments*. Machine-wide and OS health (root-kit detection, OS stuck), power management and coarse hardware resource allocation can be managed centrally while insulating partitions from harmful effects of other partitions. As with other software decomposition strategies, this simplifies software construction. Coarse partitioning also provides a good way to get coarse parallelism. Applications running concurrently in separate trust domains need the benefits of either rich operating environments or specialized environments that provide specific guarantees (such as real time scheduling). This also provides a vehicle to stage new facilities while retaining legacy environments unmodified. Each OS partition can exercise finer resource control over the resources it controls in conjunction with its application mix. Within a process, the application and supporting runtime can exert very fine grain control over resources in conjunction with the OS. Further, the OS must include a better asynchronous "system API" and lightweight native threads. Finally, the system stack must manage heterogeneous hardware; general purpose cores, GPUs, vector units and special cores such as encryption or compression.

## New applications

Can people user this much computing power? Yes.[9] The ultimate application mix is hard to forecast (applications that need this level of computing don't, by definition, exist, and the application specialists will not invest the effort required until they see some hardware). Again, we can speculate.

It is uncontroversial that servers (including home servers) will also benefit from many-core computing, and this will also boost the need for powerful clients. With cheap and ubiquitous sensors and natural language processing, we can anticipate environment aware, multi-media (vision, speech, gesture, object recognition, etc.) input and output human computer interfaces that "learn" user behaviors and offer suggestions or possibly automatically manage some tasks for users.

[9] We have a well documented historical answer if not a complete proof.

Better data mining and modeling will provide business intelligence and targeted customer service. Automated medical imaging, diagnosis and well being monitoring will be commonplace. High-level tools like MATLAB or Excel, designed for parallelism, will take advantage of increased power and delegate processing across the network, provided the right workflow tools are integrated.

With terabyte disks, these systems will make superb media library, capture, edit, and playback systems. Film fans can purchase, download, and view protected feature films on opening day. Most printed material and other media can be replaced with electronic versions, accessed via a broadband connection, with vastly improved search and cross reference capabilities. These machines can make virtual reality and realistic games, well, real. Not only entertainment but education will benefit.

Today's corporate servers will shrink to a few racks and become highly resilient to failure. State check pointing and load balancing will improve performance and reliability. Damage from catastrophic failures is limited to a few seconds of downtime and rollback. Provisioning, deploying, and administering these servers and applications are simplified and automated.

Massively parallel computational grids built of commodity hardware already solve scientific problems like computational chemistry, protein folding and drug design. "Supercomputers" already analyze nuclear events and water tables and predict the climate and the economy. The power of these systems and the reach of these techniques will vastly improve with new hardware, and scientists will have supercomputers under their desks. By the way, scientific, financial and medical "supercomputing" are no longer "small" business opportunities. More than 10% of servers are used in scientific applications.

Classic computational techniques (known in the scientific community as the "seven dwarves"[10] – including equation solvers, adaptive mesh modeling, etc.) will help explore regimes that will change our lives.[11] Already, Microsoft researchers and world class scientists are using advanced computational techniques to explore potential cures for Aids and cancer, model Hydrologic activity in agriculturally sensitive regions, perform seismic modeling and run virtual laboratories for advanced physics. As in the past, use by scientists will help illuminate the path for the rest of us.

[10] So named by Phil Collela of LBL, these include the important computational kernels for modeling and analysis.

[11] Asanovic, K.ryste et. al., "The Landscape of Parallel Computing Research: A View from Berkeley," UCB/EECS-2006-183.

## No time to waste

Programmable systems are playing an increasingly large part in our lives and, in many ways, provide a world-wide "paradigm shift" comparable to the appearance of cheap, mass market printing in scope and benefit. Many-core computers signal a shift in Computer Science, Computational Science, and classical Commercial Software that (as in all good technology shifts) marry the past advances of many "knowledge workers" as well as provide a new avenue for qualitatively new advances.

**Acknowledgements**
The author is indebted to a number of colleagues at Microsoft for insight and review of this material including Craig Mundie, Burton Smith, David Callahan, Jim Larus, Jan Gray, Paul England, Brian LaMacchia and Tony Hey.

# The Role of Multicore Processors in the Evolution of General-Purpose Computing

John McCalpin
Chuck Moore
Phil Hester
Advanced Micro Devices, Inc.

## Introduction

The title of this issue, "The Promise and Perils of the Coming Multicore Revolution and Its Impact", can be seen as a request for an interpretation of what multicore processors mean now, or as an invitation to try to understand the role of multicore processors from a variety of different directions. We choose the latter approach, arguing that, over time, multicore processors will play several key roles in an ongoing, but ultimately fundamental, transformation of the computing industry.

This article is organized as an extended discussion of the meaning of the terms "balance" and "optimization" in the context of microprocessor design. After an initial review of terminology (Section 1), the remainder of the article is arranged chronologically, showing the increasing complexity of "balance" and "optimization" over time. Section 2 is a retrospective, "why did we start making multiprocessors?" Section 3 discusses ongoing developments in system and software design related to multicore processors. Section 4 extrapolates current trends to forecast future complexity, and the article concludes with some speculations on the future potential of a maturing multicore processor technology.[1]

[1] Colwell, B. "The Art of the Possible", *Computer*, vol. 37, no. 8, August 2004, pp.8-10.

## 1. Background

### Key Concepts

Before attempting to describe the evolution of multicore computing, it is important to develop a shared understanding of nomenclature and goals. Surprisingly, many commonly used words in this area are used with quite different meanings across the spectrum of producers, purchasers, and users of computing products. Investigation of these different meanings sheds a great deal of light on the complexities and subtleties of the important trade-offs that are so fundamental to the engineering design process.

### Balance

Webster's online dictionary lists twelve definitions for "balance."[2] The two that are most relevant here are:

[2] Merriam-Webster Online: http://www.m-w.com/dictionary/balance

- **5 a :** stability produced by even distribution of weight on each side of the vertical axis **b :** equipoise between contrasting, opposing, or interacting elements **c :** equality between the totals of the two sides of an account
- **6 a :** an aesthetically pleasing integration of elements

Both of these conceptualizations appear to play a role in people's thinking about "balance" in computer systems. The former is quantitative and analytical, but attempts to apply this definition quickly run into complications. What attributes of a computer system should display an "even distribution"? Depending on the context, the attributes of importance might include cost, price, power consumption, physical size, reliability, or any of a host of (relatively) independent performance attributes. There is no obvious "right answer" in choosing which of these attributes to "balance" or how to combine multiple comparisons into a single "balanced" metric.

The aesthetic definition of "balance" strikes an emotional chord when considering the multidimensional design/configuration space of computer systems, but dropping the quantitative element eliminates the utility of the entire "balance" concept when applied to the engineering and business space.

The failure of these standard definitions points to a degree of subtlety that cannot be ignored. We will happily continue to use the word "balance" in this article, but with the reminder that the word only makes quantitative sense in the context of a clearly defined quantitative definition of the relevant metrics, including how they are composed into a single optimization problem.

### Optimization

Webster's online dictionary lists only a single definition for "optimization":[3]

- **:** an act, process, or methodology of making something (as a design, system, or decision) as fully perfect, functional, or effective as possible; *specifically* : the mathematical procedures (as finding the maximum of a function) involved in this

[3] Merriam-Webster Online: http://www.m-w.com/dictionary/optimization

For this term, the most common usage error among computer users draws from the misleadingly named "optimizers" associated with compilers. As a consequence of this association, the word "optimization" is often used as a synonym for "improvement." While the terms are similar, optimization (in its mathematical sense) refers to minimizing or maximizing a particular objective function in the presence of constraints (perhaps only those provided by the objective function itself). In contrast, "improvement" means to "make something better" and does not convey the sense of trade-off that is fundamental to the concept of an "optimization" problem. Optimization of a computer system design means choosing parameters that minimize or maximize a particular objective function, while typically delivering a "sub-optimal" solution for other objective functions. Many parameters act in mutually opposing directions for commonly used objective functions, e.g., low cost vs high performance, low power vs high performance, etc. Whenever there are design parameters that act in opposing directions, the "optimum" design point will depend on the specific quantitative formulation of the objective function – just exactly how much is that extra 10% in performance (or 10% reduction in power consumption, etc.) worth in terms of the other design goals?

### Methodology

For the purposes of this article, the performance impacts of various configuration options will be estimated using an analytical model with coefficients "tuned" to provide the best fit for a large subset of the SPECfp2000 and SPECfp_rate2000 benchmarks collected in March 2006.[4] The analysis included 508 SPECfp2000 results and 730 SPECfp_rate2000 results. The remaining 233 results were excluded from the analysis because either advanced compiler optimizations or unusual hardware configurations made the results inappropriate for comparison with the bulk of the results.

[4] All data is freely available at: http://www.spec.org/cpu2000/

The performance model has been described previously[5] [6] but has been extended here to include a much more complete set of data and has been applied to each of the 14 SPECfp2000 benchmarks as well as to the geometric mean values. Although the model does not capture some of the details of the performance characteristics of these benchmarks, using a least-squares fit to a large number of results provides a large reduction in the random "noise" associated with the individual results and provides a significant degree of platform-independence.

[5] McCalpin, J. "Composite Metrics for System Throughput in HPC", unpublished work presented at SuperComputing2003, http://www.cs.virginia.edu/~mccalpin/SimpleCompositeMetrics2003-12-08.pdf

[6] McCalpin, J. "Composite Metrics for System Throughput in HPC", presented at the HPC Challenge Benchmark panel discussion at SuperComputing2003, http://www.cs.virginia.edu/~mccalpin/CompositeMetricsPanel2003-11-20b.pdf

In brief, the model assumes that the execution time of each benchmark is the sum of a "CPU time" and a "Memory Time," where the amount of "work" done by the memory subsystem is a simple function of the cache size – linearly decreasing from a maximum value with no cache to a minimum value with a "large" cache (where "large" is also a parameter of the model), and a constant amount

of memory work for caches larger than the large size. The rate at which CPU work is completed is assumed to be proportional to the peak floating-point performance of the chip for 64-bit IEEE arithmetic, while the rate at which memory work is completed is assumed to be proportional to the performance of the system on the 171.swim (base) benchmark. Previous studies have shown a strong correlation between the performance of the 171.swim benchmark and direct measurement of sustained memory performance using the STREAM benchmark.[7]

[7] McCalpin, J. "Using Timings for the SWIM Benchmark from the SPEC CPU Suites to Estimate Sustainable Memory Bandwidth", revised to 2002-12-02, http://www.cs.virginia.edu/stream/SWIM-BW.html

The model results are strongly correlated with the measured results, with 75% of the measurements falling within 15% of the projection. This suggests that the underlying model assumptions are reasonably consistent with the actual performance characteristics of these systems on these benchmarks. Although there are some indications of systematic errors in the model, not all of the differences between the model and the observations are due to oversimplification of the hardware assumptions – much of the variance also appears to be due to differences in compilers, compiler options, operating systems, and benchmark configurations. Overall, the model seems appropriately robust to use as a basis for illustrations of performance and price/performance sensitivities in microprocessor-based systems.

### Assumptions and Modeling

For the performance and price/performance analysis, we will assume

- The bare, two-socket system (with disks, memory, and network interfaces, but without CPUs) costs $1,500.
- The base CPU configuration is a single-core processor at 2.4 GHz with a 1 MB L2 cache, costing $300.
- The die is assumed to be about ½ CPU core and about ½ L2 cache, with the other on-die functionality is limited to a small fraction of the total area.
- The "smaller chip" configuration is a single-core processor at 2.8 GHz with a 1 MB L2 cache, costing $150.
- The "lots of cache" configuration is a single-core processor at 2.8 GHz with a 3 MB L2 cache, costing $300.
- The "more cores" configuration is a dual-core processor at 2.0 GHz with 1 MB L2 cache per core, costing $300.

## 2. Initial Development of Multicore Chips

The initial development of multicore processors owed much to the continued shrinking of CMOS lithography. It has long been known that increasing the size/width of a CPU core quickly reaches the realm of diminishing returns.[8] So as the size of a state-of-the-art core shrinks to a small fraction of an economically viable die size, the options are:

[8] Farkas, K. I., Jouppi, N. P., Chow, P.. "Register file design considerations in dynamically scheduled processors", 1996. *Proceedings: Second International Symposium on High-Performance Computer Architecture*, pp.40-51, 3-7 Feb 1996.

- Produce a smaller chip
- Add lots of cache
- Add more cores

The option of adding more memory bandwidth typically adds significant cost outside of the processor chip, including revised motherboards (perhaps requiring more layers), additional DIMMs, etc. Because of these additional costs and the burden of socket incompatibility, the option of adding more memory bandwidth will be considered separately from options that involve only processor die size.

Figures 1 and 2 show some of the relative performance and performance/price metrics for these three options, assuming a 30% lithography shrink (i.e., 50% area shrink) which also allows a 17% frequency increase for the single core, but requires a 17% frequency decrease for the dual-core (to remain in the same power envelope). Note that the SPECfp_rate2000 benchmark consists of 14 individual tests that are modeled independently. These are summarized as a *minimum speedup*, *median speedup*, *geometric mean speedup*, and *maximum speedup*. For the dual-core processor option, both the single-core (uni) and dual-core (mp) speedups are estimated.

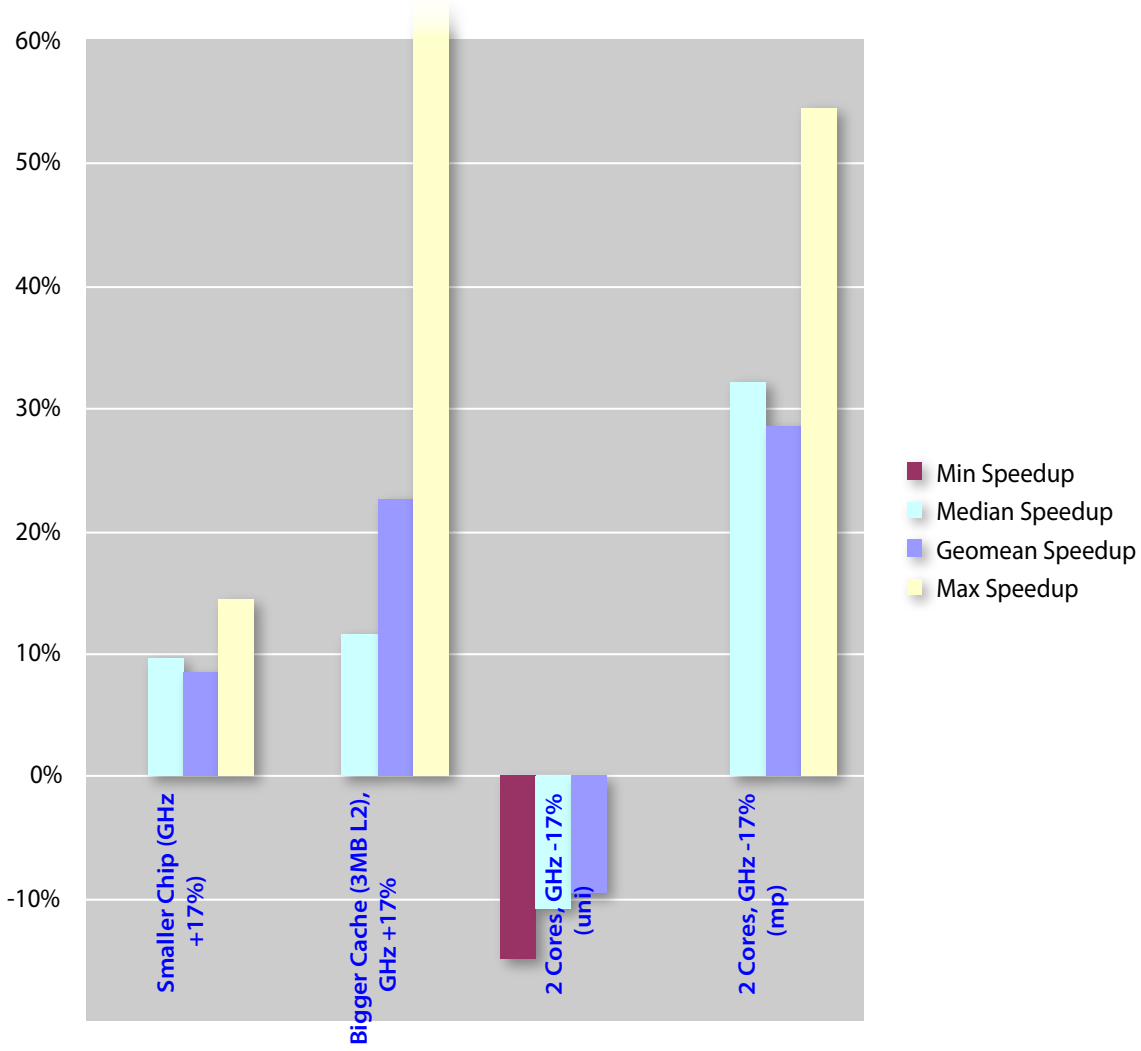


Figure 1: Estimated Relative SPECfp_rate2000 performance for three system configurations (based on the analytical model described in Section 1), assuming a 30% lithography shrink (50% area reduction) with a corresponding 17% frequency boost for the single-core chip and a 17% frequency reduction for the dual-core chip. The *Smaller Chip* is ½ the size of the reference chip, while the *Bigger Cache* and *2 Core* versions are scaled to be the same size and have the same power requirements as the reference chip. Note that the Max Speedup for the *Bigger Cache* case is +156%, but is truncated by the scale here.

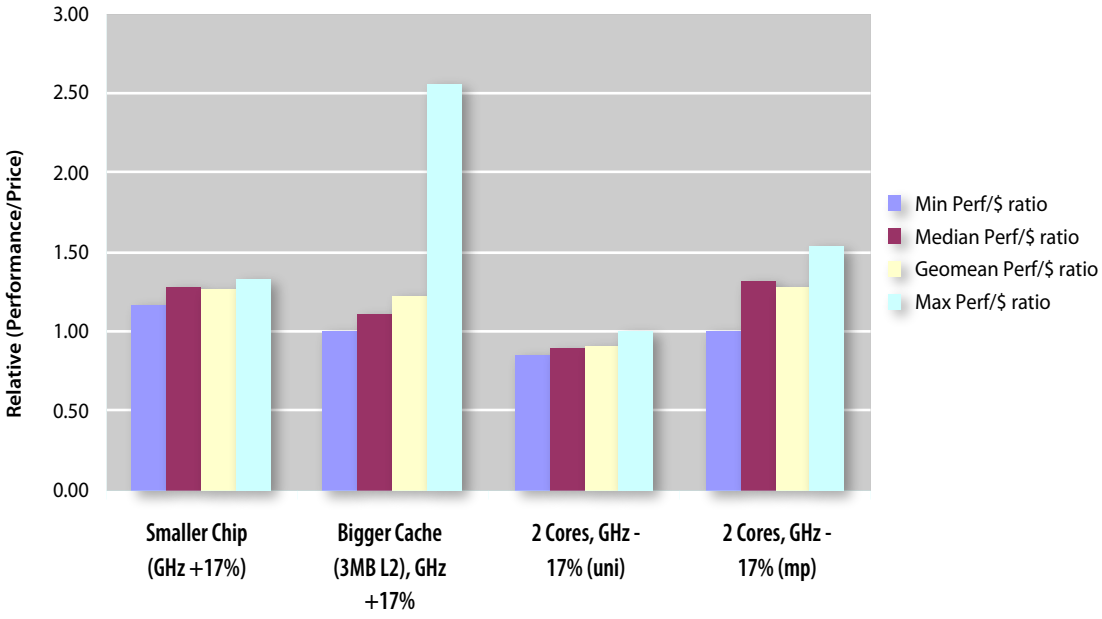


Figure 2: Estimated Relative SPECfp_rate2000/price for three system configurations (based on the analytical model described in Section 1), assuming a 30% lithography shrink (50% area shrink) with a corresponding 17% frequency boost for the single-core chip and a 17% frequency reduction for the dual-core chip.

Looking at these three options in turn:

### Produce Smaller Chips

Shipping smaller/cheaper chips with modest frequency increases is of modest value in improving performance and performance/price for customers. In this example, halving the price of the processor reduces the cost of the overall system by 14% ($1,800 vs $2,100) while the 17% frequency boost provides from 0% to 14% improvement in performance, with median and geometric mean improvements of 8%-9% (Figure 1). The combination of these two factors provides performance/price improvements of 17% to 33% with median and geometric mean performance/price improvements of 27%-28%.

### Add Lots of On-Chip Cache

Adding lots of cache provides more variable improvement across workloads than most other options. In this example, tripling the L2 cache from 1 MB to 3 MB provides performance improvements of 0% to 127%, with a median improvement of 0% and an improvement in the geometric mean of 11.8%.

The 17% CPU frequency improvement associated with the cache size improvement provides additional benefits; with the combined performance improvement ranging from 0% to 156%, a median improvement of 11.5%, and an improvement in the geometric mean of 22.5%.

In this case, the assumed cost of the chips is the same as the reference system, so the performance/price shows the same ratios as the raw performance.

### Add CPU Cores

Adding cores improves the throughput across a broad range of workloads, at the cost of a modest (17%) reduction in frequency in order to meet power/cooling restrictions. Here we assume that the 50% area reduction allows us to include two CPU cores, each with the same 1 MB L2 cache as the reference chip and at the same cost. When running a single process, the performance varies from 15% slower than the reference platform to 0% slower, with median and geometric mean changes of -10% to -11%.

If we can use the second core to run a second copy of the code, the throughput of the system increases from a minimum of 0% to a maximum of 54%, with median and geometric mean speedups of 29% to 32%.

In this case, the assumed cost of the chips is the same as in the reference system, so the performance/price shows the same ratios as the raw performance.

### Discussion

The three examples above provided a disturbingly large number of independent performance and performance/price metrics – 70 relative values. Reducing the 14 performance values per benchmark to three (minimum, geometric mean, maximum) still leaves us with nine performance values and 12 performance/price values (of which nine are identical to the performance values). Combining these into a metric that can be used to make rational design decisions is not a trivial task.

Each of the three options has significant benefits and significant disadvantages:

| Design Option | Major Benefits | Major Drawbacks |
|---|---|---|
| Small Chip | Price reduction | Weakest best-case improvements |
| Big Cache | Huge performance boost on a few codes | Weakest median and geometric mean performance/price improvements |
| Dual-Core | Strongest median and geometric mean throughput improvements | Decreased single-processor performance |

It is relatively straightforward to find customers for whom any of the six *Major Benefits* or *Major Drawbacks* constitutes critical decision factors. It is much more complex to determine how to take that information, generalize it, and use it in support of the company's business model.

Engineering decisions must, of course, support the business model of the company making the investments. Even the seemingly simple goal of "making money" becomes quite fuzzy on detailed inspection. Business models can be designed to optimize near-term revenue, near-term profit, long-term revenue, long-term profit, market "buzz" and/or "goodwill", or they can be designed to attain specific goals of market share percentage or to maximize financial pressure on a competitor. Real business models are a complex combination of these goals, and unfortunately for the "purity" of the engineering optimization process, different business goals can change the relative importance of the various performance and performance/price metrics.

### Caveat

In each of these scenarios the performance changes depend on the ratio of memory performance to CPU performance on the baseline system. As the amount of available bandwidth increases, the benefits of larger caches decrease and the benefit of having more CPU cores increases. Conversely,

relatively low memory bandwidth makes large caches critical but significantly reduces the throughput improvements obtainable with additional CPU cores.

For the more cache-friendly SPECint_rate2000 benchmark, results on the IBM e326 servers running at 2.2 GHz show that doubling the number of cores per chip at the same frequency results in a throughput improvement of 65% to 100% on SPECint_rate2000 (geometric mean improvement = 95%).[9]

[9] Results published at http://www.spec.org/ with IBM e326 2-socket servers running single-core and dual-core parts at 2.2 GHz, all running Suse Linux 9.0 and version 2.1 of the PathScale EKOPath compiler suite.

## 3. Current and Near-Term Issues

### The "Power Problem"

Like performance, power is more multidimensional than one might initially assume. In the context of high-performance microprocessor-based systems, the "power problem" might mean any of:

- Bulk current delivery to the chip through a large number of very tiny pins/pads. (Note that as voltages go down, currents go up and resistive heating in the pins/pads increases even at the same bulk power level.)
- Bulk heat removal to prevent die temperature from exceeding threshold values associated with significantly shortened product lifetime.
- "Hot Spot" power density in small areas of the chip that can cause localized failures. (Note that halving the size of the processor core on the die while increasing frequency to maintain the same bulk power envelope causes a doubling of power density within the core.)
- Bulk power deliver to the facility housing the service – including cost of electricity plus the cost of electrical upgrades.
- Bulk heat removal from the facility housing the servers – including the cost of electricity plus the cost of site cooling system upgrades.
- Bulk heat removal from the processor chips that preheats air flowing over other thermally sensitive components (memory, disk drives, etc.).

So the power problem actually refers to at least a half-dozen inter-related, but distinct, technical and economic issues.

### Throughput vs Power/Core vs Number of Cores

Restricting our attention to the issue of delivering increased performance at a constant power envelope, we can discuss the issues associated with using more and more cores to deliver increased throughput.

Depending on exactly what parameters are held constant, power consumption typically rises as the square or cube of the CPU frequency, while performance increases less than linearly with frequency. For workloads that can take advantage of multiple threads, multiple cores provide significantly better throughput per watt. As we saw earlier however, this increased throughput comes at the cost of reduced single-thread performance.

In addition to waiting for lithography to shrink so that we can fit more of our current cores on a chip, the opportunity also exists to create smaller CPU cores that are intrinsically smaller and more energy efficient. This has not been common in the recent past (with the exception of the Sun T1

processor chips) because of the assumption that single-thread performance was too important to sacrifice. We will come back to that issue in the section on *Longer-Term Extrapolations*, but a direct application of the performance model above shows that as long as the power consumption of the CPU cores drops faster than the peak throughput, the *optimum* throughput will always be obtained by an infinite number of infinitesimally fast cores. Obviously, such a system would suffer from infinitesimal single-thread performance, which points to a shortcoming of optimizing to a single figure of merit. In response to this conundrum, one line of reasoning would define a *minimum acceptable single-thread performance* and then optimize the chip to include as many of those cores as possible, subject to area and bulk power constraints.

There are other reasons to limit the number of cores to a modest number. One factor, which is missing in the simple throughput models used here, is communication and synchronization. If one desires to use more than one core in a single parallel/threaded application, some type of communication/synchronization will be required, and it is essentially guaranteed that for a fixed workload, the overhead of that communication/synchronization will be a monotonically increasing function of the number of CPU cores employed in the job.

This leads to a simple modification of Amdahl's Law:

$$T = T_s + \frac{T_p}{N} + T_o \times N$$

Here $T$ is the total time to solution, $T_s$ is the time required to do serial (non-overlapped) work, $T_p$ is the time required to do all of the parallelizable work, $N$ is the number of processors applied to the parallel work, and $T_o$ is the overhead per processor associated with the communication & synchronization required by the implementation of the application. This last term representing the growth in overhead as more processors are added is the addition to the traditional Amdahl's Law formulation.

In the standard model (no overhead), the total time to solution is a monotonically decreasing function of $N$, asymptotically approaching $T_s$. In the modified formation, it is easy to see that as $N$ increases there will come a point at which the total time to solution will begin to increase due to the communication overheads. In the simple example above, a completely parallel application ($T_s$=0) will have an optimal number of processors defined by:

$$N_{opt} = \sqrt{\frac{T_p}{T_o}}$$

So, for example, if $T_o$ is 1% of $T_p$, then maximum performance will be obtained using 10 processors. This may or may not be an interesting design point, depending on the relative importance of other performance and performance/price metrics.

### Market Issues

There are intriguing similarities between multi-core chips of the near future and the RISC SMPs that allowed the development of the RISC server market in the 1990's -- a market responsible for over $240B in hardware revenue in the last 10 years.[10]

Like the RISC SMPs of the mid-1990's, these multi-core processors will provide an easy-to-use, cache-coherent, shared-memory model, now shrunk to a single chip.

[10] All revenue data here is based on an AMD analysis of the IDC 2Q2006 world-wide server tracker report. http://www.idc.com/

In ~1995, the SGI POWER Challenge was the best selling HPC server in the mid-range market – one of us (McCalpin) bought an 8-cpu system that year for about $400,000. CPU frequency in 1996-1997 was 90 MHz (11 ns) with a main memory latency close to 1000 ns, or 90 clock periods. In 2007, a quad-core AMD processor will have a frequency of over 2 GHz (0.5 ns) with a main memory latency of ~55 ns, or 110 clock periods. These are remarkably similar ratios.

Delivering adequate memory bandwidth was a challenge (sorry for the pun) on the RISC SMPs. An 8-cpu SGI POWER Challenge had a peak floating-point performance of 2.88 GFLOPS with a peak memory bandwidth of 1.2 GB/s, about 0.42 Bytes/FLOP. A quad-core AMD processor will launch with peak floating-point performance of about 32 GFLOPS and a peak memory bandwidth of about 12.8 GB/s, also giving about 0.4 Bytes/FLOP.

By 1996, the UNIX server market was generating over $22B in hardware revenue, increasing to almost $33B in 2000. The market has been in decline since then, dropping to about $18B in 2006.

Three factors have combined to lead to this decline:

- Increasing difficulty in maintaining the system balances that initially made the servers successful,
- Inability for larger RISC SMPs to follow the smaller RISC SMPs to lower price per processor, and
- Introduction of even less expensive servers based on the IA32 architecture, accelerating with the introduction of products based on the AMD64 architecture in 2003.

It is interesting to look at these three factors in more detail.

### Shifting System Balance

As noted above, the initial RISC SMPs had main memory latency in the range of 100 CPU clocks and bandwidth in the 0.4 Bytes/FLOP range. The latency was largely independent of CPU count, while the bandwidth per processor could be adjusted by configuring different numbers of processors.

There has been a clear systematic correlation between application area and bandwidth per processor, with "cache-friendly" application areas loading up the SMPs with a full load of processors, and "high-bandwidth" areas configuring fewer processors or sticking with uniprocessor systems.

By the year 2000, main memory latencies in RISC SMPs had decreased by a factor of about three, while CPU frequencies had increased by factors of three to six. Bandwidth per processor became more complex as monolithic system buses shifted to a variety of NUMA implementations.

### Price Trends

During the second half of the 1990's, server vendors went to great lengths to maintain the desirable system balance properties of their very successful systems from the early to middle 1990's. Although largely successful, this effort had a cost -- a financial cost. The two major contributors to this cost were off-chip SRAM caches and snooping system buses for cache coherency. The large, off-chip SRAM caches were critical for these systems to tolerate the relatively high memory latencies and to reduce the bandwidth demand on the shared address and data buses. When Intel quit using standard, off-chip SRAM caches, the market stalled and price/performance of SRAM failed to follow the downward trend of other electronic components. By the year 2000, a large, off-chip SRAM cache could cost several times what the processor cost.

For small SMPs, however, the reduced sharing of the memory and address bus meant lower latency and higher bandwidth per processor. These, in turn, allowed the use of smaller off-chip SRAM caches. The gap between price/processor of small RISC SMPs and large RISC SMPs widened, and customers increasingly turned to clusters of small SMPs instead of large SMPs.

### Killer Micros

By the early 2000's, servers based on commodity, high-volume architectures had come within striking distance of the absolute performance of servers based on proprietary RISC architectures, with the high-volume servers delivering superior price/performance. The trend toward small RISC SMPs made the transition to small commodity SMPs much easier. This trend was given a large boost in 2003 with the introduction of processors based on the AMD64 architecture, providing even better performance and native 64-bit addressing and integer arithmetic. Intel followed with the EM64T architecture, leading to a remarkably non-disruptive transition of the majority of the x86 server business from 32-bit to 64-bit hardware in just a few years.

These trends should not be read as indicating a lack of customer interest in SMPs. They do, however, provide an indication of the price sensitivity of various customers to the capabilities provided by the larger SMP systems. Make the price difference too large, and the market will figure out how to use the cheaper hardware.

Just as the RISC SMP market resulted in the parallelization of a large number of ISV codes (in both enterprise and technical computing), the multicore processor trend is likely to generate the impetus to parallelization of the much larger software base associated with the dramatically lower price points of today's small servers.

Unlike the RISC SMP market of the 1990's, the multicore processors of today do not rely on off-chip SRAM caches and can be configured to avoid expensive chip-to-chip coherence traffic (either through snoop filters or by simply using single-chip servers, e.g., Sun's T1/Niagara). There is no obvious general-purpose competitor overtaking x86 performance from lower price points, except perhaps in the case of mobile/low-power devices.

## 4. Longer-Term Extrapolations

### SMP on a chip

The RISC SMP market of the mid to late 1990's was dominated by 4, 8, and 16-way SMPs. This "sweet spot" in the market provided enough extra CPU power to justify efforts at parallelizing applications, without incurring unacceptable price overheads or providing too many processors for applications to use effectively. Current trends suggest that similar SMP sizes will be available on a single chip within a few years, leading one to speculate in several directions:

- Will multicore chips create a new market for parallel codes in the same way that RISC SMPs created the UNIX server market in the 1990's?
- Will effective exploitation of multicore processor chips require fundamental changes in architecture or programming models, or will growth in parallel applications happen on its own – perhaps supported by more modest architectural enhancements (e.g., transactional memory)?
- Will the increasing number of cores/threads available on a single chip obviate the need for larger SMP systems for the majority of users?

Even at the scale of one socket and two socket systems, the increasing number of cores per chip will likely lead to users running combinations of multithreaded and single-threaded jobs (none using all the CPU cores) – more like the large SMP servers of the last decade than like traditional usage models. Increasing numbers of cores is also likely to lead to widespread adoption of virtualization even in these small systems, with multiple guest operating systems having dedicated cores, but competing for memory space, memory bandwidth, shared caches, and other shared resources.

### Design Space Explosion

The simple examples at the beginning of this article demonstrated the complexity of defining appropriate figures of merit for microprocessor chips with fairly limited degrees of freedom (e.g., one or two cores plus small or large cache). Dual-core processors first shipped in volume from 90 nm fabrication processes, with quad-core processors expected in quantity in 2007 based on 65 nm process technology. Going to 45 nm provides the possibility for another doubling (8 cores), 32 nm provides the possibility for another doubling (16 cores), and 22 nm technology providing yet another doubling (32 cores) is considered feasible.

Recent studies[11] have shown that the CMP design space is multidimensional both from the engineering and application performance perspectives. The parameter space associated with all the design possibilities inherent in the flexibility of having so many independent "modules" on chip is huge, but the real problem is that the definition of performance and performance/price metrics grows to have almost as many degrees of freedom. If every application of interest has different optimal design points for single-threaded performance, multi-threaded performance design point, single-threaded performance/price design point, and multi-threaded performance/price, then making design decisions is going to become even more difficult, and the struggle between commodity volumes and a proliferation of independent "optimal" design points will become one of the key challenges facing the industry.

[11] Li, Y., et al. "CMP Design Space Exploration Subject to Physical Constraints", *The Twelfth International Symposium on High-Performance Computer Architecture*, 2006, 11-15 Feb, 2006, pp 17-28. http://www.cs.virginia.edu/papers/hpca06.pdf

### Heterogeneous Architectures

It is straightforward to demonstrate that, assuming different resources have different costs, a homogeneous multicore chip cannot be optimal for a heterogeneous workload. Expansion of the design space to include heterogeneous processor cores adds many new degrees of freedom. Parameters that might vary across cores include:

- Base ISA
- ISA extensions
- Cache size(s)
- Frequency
- Issue width
- Out of Order capability

The myriad possibilities created by multiplying the homogeneous multicore design space by this extra set of degrees of freedom are both exciting and daunting.

Of course multicore processors will not be limited to containing CPUs. Given the widespread incorporation of three-dimensional graphics into mobile, client, and workstation systems, integration of graphics processing units (or portions of graphics processing units) onto the processor chip seems a natural evolution, as in AMD's announced "Fusion" initiative. Other types of non-CPU heterogeneous architectures may become logical choices in the future, but today none of the contenders appear to have the critical mass required to justify inclusion in high-volume processor products.

### Too Many Cores

While the short-term expectation of 4-8 CPU cores per chip is exciting, the longer-term prospect of 32, 64, 128, 256 cores per chip raises additional challenging issues. During the peak of the RISC SMP market in the late 1990's, large (8p – 64p) systems were expensive and almost always shared. Individual users seldom had to worry about finding enough work to do to keep the CPUs busy. In contrast, the multicore processor chips of the near future will be inexpensive commodity products. An individual will be able to easily afford far more CPU cores than can easily be occupied using traditional "task parallelism" (occupying the CPUs running independent single-threaded jobs). In 2004, for example, a heavily-configured 2-socket server based on AMD or Intel single-core processors typically sold for $5,000 to $6,000, allowing a scientist/engineer with a $50,000 budget to purchase approximately eight servers (16 cores) plus adequate storage and networking equipment. Since the explosion in popularity of such systems beginning around the year 2000, many users have found that these small clusters could be fully occupied running independent serial jobs, or jobs parallelized only within a single server (using OpenMP or explicit threads). Assuming that vendors deliver at approximately the same price points, 16-core processor chips will provide 256 cores for the same budget. Few scientific/engineering users have such large numbers of independent jobs (typically generated by parameter surveys, sensitivity analyses, or statistical ensembles) and do not consider such throughput improvements a substitute for improving single-job performance. Clearly, the prospect of a 128-core chip providing 2048 threads for a $50,000 budget calls for a fundamental change in the way most users think about how to program and use computers. There is therefore a huge burden on the developers of multicore processors to make it much easier to exploit the multiple cores to accelerate single jobs as well as a huge opportunity for computing users to gain competitive advantage if they are able to exploit this parallelism ahead of their competition.

### What about Bandwidth?

The flexibility allowed by multicore implementations, along with the quadratic or cubic reduction of power consumption with frequency, will allow the development of processor chips with dramatically more compute capability than current chips. Sustainable memory bandwidth, on the other hand, is improving at a significantly slower rate. DRAM technology has improved its performance primarily through increased pipelining and this approach is reaching its practical limits. Power consumption associated with driving DRAM commands and data, consuming data from the DRAMs, and transmitting/receiving data and probe/snoop traffic to/from other chips are growing to be non-trivial components of the bulk power usage. As noted above, historical data suggests that systems supporting less than about 0.5 GB/s main memory bandwidth per GFLOP/s of peak floating point capability are significantly less successful in the marketplace. Looking forward to, for example, eight cores with four floating point operations per cycle per core at 3 GHz, it is clear that 100 GFLOPS (peak) chips are not a distant dream but a logical medium-term extrapolation. On the other hand, sustaining 50 GB/s of memory bandwidth per processor chip looks extremely expensive. Even if DDR2/3 DRAM technology offers data transfer rates of 1600 MHz (12.8 GB/s per 64-bit channel), actually sustaining the desired levels of memory bandwidth will require many channels (probably eight channels for 102.4 GB/s peak bandwidth), meaning at least eight DIMMs (which can easily be more memory than desired), and will require on the order of 40 outstanding cache misses to reach 50% utilization. (Assuming 50 ns memory latency, the latency-bandwidth product for 102.4 GB/s is 5120 Bytes, or 80 cache lines at 64 Bytes each. So it will require approximately 40 concurrent cache line transfers to reach the target 50 GB/s sustained bandwidth.)

## Summary & Conclusions

The examples and illustrations in this article suggest that we are just beginning to grapple with the many opportunities and challenges associated with multicore processors. Our initial steps have been limited by technology to modest changes in the overall balance of systems, but technology trends make it clear that the flexibility provided by future process technologies will present us with an abundance of opportunities to design microprocessor-based systems with radically different power, performance, and cost characteristics. The tension between maintaining high volumes by selling standardized products and increasing performance, performance/watt and performance/price by creating multiple differentiated products will be a major challenge for the computing industry. Even if we try to maintain a modest number of "fast" cores, process technology will enable us to provide more cores than users are currently capable of using effectively. This will challenge industry, academia, and computer users to work together to develop new methods to enable the use of multiple cores for "typical" applications, exploiting the physical locality of on-chip communications to enable more tightly coupled parallelism than has previously been widely adopted.

# High Performance Computing and the Implications of Multi-core Architectures

Dave Turek
IBM

Over the past several years, public sector institutions (universities and government) and commercial enterprises have deployed supercomputing systems at an unparalleled rate, courtesy of favorable acquisition economics and compelling technological and business process innovation.

Manufacturers and producers of supercomputers have leveraged the price performance improvements of commodity microprocessors, the innovation in interconnect technologies, and the rise of Linux and open source software to reach classes of consumers unheard of as recently as five years ago.

In June of 1997, the so-called "fastest computer in the world," the ASCI Red machine at Sandia National Laboratory, was the first system exceeding a teraflop in compute power.[1] Today, the same amount of compute power could be acquired for around $200,000, making supercomputing affordable to small companies, single academic departments and, in some cases, even individual researchers.

[1] Top500: http://www.top500.org/

While these dramatic improvements in systems affordability have been taking place, the ability to extract value from the system is principally the consequence of well written and effective software. Here, the story for the industry is not quite as sanguine. From our customers, we hear that these systems are still difficult to use (for complete exploitation), and the applications they need either need to be ported, or even rewritten, to properly take advantage of all the hardware innovation in modern supercomputers.

This view is universal and strikes at the heart of the economic or scientific competitiveness of the institution: "Today's computational science ecosystem is unbalanced, with a software base that is inadequate to keep pace with and support evolving hardware and application needs … The result is greatly diminished productivity for both researchers and computing systems."[2]

[2] "Computational Science: Ensuring America's Competitiveness," President's Information Technology Advisory Committee, June 2005.

As we contemplate each new hardware innovation for supercomputing, we must understand at the most fundamental level that software is the key to unlocking the value of the system for the benefit of the enterprise or the researcher.

## Moore's Law and Multicore CPUs in Supercomputing

The dramatic increase in the deployment and utilization of supercomputing owes much to microprocessor innovation and the attendant improvement in price performance typically associated with improved transistor density. But the "simple" idea of Moore's Law, characterized as the doubling transistor density per unit time, has its limits. Just as one cannot fold a piece of paper in half and in half again without reaching a fundamental limit within just a few folds, neither is it reasonable to think that increased density of transistors can be pursued without reaching some rather material physical limits over time.

One reaction to this reality has been the emergence of multi-core CPUs, each core running at a somewhat modest (but still significant) frequency, but orchestrated to work in concert on the computational problem at hand. This approach is meant to finesse the design limitations of forever-faster, single core CPUs while still attending to the insatiable needs for compute power in all market segments. And the supercomputing market segment, having more need for speed than any other segment, is aggressive in its pursuit and acceptance of this innovation.

The implications of multi-core designs to supercomputing are profound in terms of overall benefit. The fastest supercomputer in the world today, the Blue Gene/L system from IBM installed at Lawrence Livermore National Laboratory,[1] is an innovative, multi-core- designed system on a chip (with each core running at less than a GhZ) that is roughly three times faster than the next fastest machine on the TOP500 list and roughly eight times faster than the NEC Earth Simulator (which was the last non-IBM system to lead the TOP500 list).

It is instructive to note that measuring from an electrical power consumption perspective, the Blue Gene system is rated on one measure at 112 MFlop/watt and the Earth Simulator at 3MFlop/Watt.[3] Even allowing for some ambiguity in measurement technique, this is a stunning difference in the power consumption characteristics of contemporary supercomputer designs and a benefit related directly to the multi-core design attributes of the Blue Gene system. Given that the fastest systems in the world today routinely consume a megawatt or more of power annually, this kind of efficiency materializes itself as very significant operational cost savings.

[3] The Green500 List: http://www.green500.org/

Perhaps one of the most ambitious multi-core designs currently available is the heterogeneous, multi-core Cell Broadband Engine (Cell BE), designed by a partnership of Sony, Toshiba and IBM and found in today's Sony PlayStation 3. This chip is currently one of the key technical components of the Roadrunner supercomputer system that IBM is building in collaboration with the Department of Energy's Los Alamos National Laboratory. This system, when complete, will be capable of sustained performance of one petaflop.

The architecture of the Roadrunner system is an example of what we expect to be a plethora of hybrid supercomputer designs that amalgamate a variety of technologies to achieve maximum performance for given applications.

## Software and Multicore Supercomputers

With the exception of government laboratories, universities, and a handful of aggressive and capable industrial companies, most software applications and tools are supplied by independent software vendors (ISVs). However, more than a third of these companies are very small (revenue less than $5M) and have limited resources to apply to the latest hardware innovations.[4]

[4] Joseph, E. "The Council on Competitiveness Study of ISVs Serving the High Performance Computing Market," *IDC Whitepaper*, http://www.compete.org/hpc

In some cases, skills to develop new algorithms that map to new hardware are in short supply, and an ISV will be more focused on serving his or her current install base of customers than in reaching out to new technologies. Also in some cases, the new hardware may be viewed as "unproven" so the ISV will wait until there is greater market acceptance before porting the application codes to the new platform. This is not a new phenomenon: "many applications for supercomputing only scale to 32 processors and some of the most important only scale to four."[5]

[5] Joseph, E., ibid.

The current Blue Gene system at Lawrence Livermore National Laboratory has 131,000 processors and the last machine on the TOP500 list, #500, has 800 processors. Furthermore, it is not uncommon for ISVs to charge license fees in proportion to the number of processors on a system (note, for the most part they do not charge license fees in proportion to the number of transistors on a chip even though more transistors and more cores attack the same object: produce more speed) giving some supercomputer customers serious sticker shock when they get their software bill.

Ultimately, the marketplace should work to resolve these issues, but they will remain serious issues for some time to come unless we see an entrepreneurial move to disturb the status quo with innovative algorithms, software and business practices that map to the capabilities of state of the art supercomputers. In the meantime, progress will continue to be made through collaborations such

as the Blue Gene Consortium, the IBM-Los Alamos Partnership on Roadrunner, and the Terra Soft HPC Collaboration around the Cell BE.

## The Benefits are Worth the Effort

While the evolution of software to better exploit multi-core architectures will unfold over time, there is a huge benefit that customers will reap and are reaping from multi-core systems today. In areas as diverse as digital media, financial services, information based medicine, oil and gas production, nanotechnology, automotive design, life sciences, material sciences, astronomy and mathematics, supercomputers have been deployed to amazing effect with material impact on the daily lives of everyone on the planet.

The ambition to get to a petaflop of computing is near universal with major efforts going on in the US, Europe and Asia for major supercomputer acquisitions in the next few years. By itself, this ambition should go a long way towards providing the stimulus to close the hardware-software gap we witness today.

CTWatch QUARTERLY

ISSN 1555-9874 VOLUME 3 NUMBER 1 **FEBRUARY 2007**

# THE PROMISE AND PERILS OF THE COMING MULTICORE REVOLUTION AND ITS IMPACT

GUEST EDITOR
**JACK DONGARRA**

AVAILABLE ON-LINE:
http://www.ctwatch.org/quarterly/

E-MAIL CTWatch QUARTERLY:
quarterly@ctwatch.org

**CTWATCH IS A COLLABORATIVE EFFORT**

http://icl.cs.utk.edu/

http://www.ncsa.uiuc.edu/

http://www.sdsc.edu/

**CTWATCH IS A PUBLICATION OF THE CYBERINFRASTRUCTURE PARTNERSHIP**

CyberInfrastructure Partnership
http://www.ci-partnership.org/

**SPONSORED BY**