CTWatch QUARTERLY February 2007

The Impact of Multicore on Computational Science Software

Jack Dongarra, Oak Ridge National Laboratory; University of Tennessee
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: *software 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]

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.

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 means 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, 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 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 particularly, 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 iteration. 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 εN 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 α 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 α 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 α 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 allow 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 allow 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.

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 twenty 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.

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 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.

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.

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

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.

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/

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.

URL to article:
http://www.ctwatch.org/quarterly/articles/2007/02/the-impact-of-multicore-on-computational-science-software/