The Impact of Multicore on Math Software[⋆]

Alfredo Buttari[1], Jack Dongarra[1,2], Jakub Kurzak[1], Julien Langou[3], Piotr Luszczek[1], and Stanimire Tomov[1]

[1] Innovative Computing Laboratory, University of Tennessee Knoxville, TN 37996, USA

[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge National Laboratory, TN 37831, USA

dongarra@cs.utk.edu

[3] Department of Mathematical Sciences, University of Colorado at Denver and Health Sciences Center, Campus Box 170, P.O. Box 173364, CO 80217-3364, USA

Abstract. Power consumption and heat dissipation issues are pushing the microprocessors industry towards multicore design patterns. Given the cubic dependence between core frequency and power consumption, multicore technologies leverage the idea that doubling the number of cores and halving the cores frequency gives roughly the same performance reducing the power consumption by a factor of four. With the number of cores on multicore chips expected to reach tens in a few years, efficient implementations of numerical libraries using shared memory programming models is of high interest. The current message passing paradigm used in ScaLAPACK and elsewhere introduces unnecessary memory overhead and memory copy operations, which degrade performance, along with the making it harder to schedule operations that could be done in parallel. Limiting the use of shared memory to fork-join parallelism (perhaps with OpenMP) or to its use within the BLAS does not address all these issues.

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 experiment back to Galileo in the early seventeenth century. Yet for all the incredible successes it enjoyed over its first three centuries, and the considerable contributions from many outstanding scientists such as G. Mendel or C. R. Darwin, the experimental

[⋆] This work was supported in part by the National Science Foundation and Department of Energy.

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 twentieth 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 (physicists, chemists, biologists, etc), 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 effort, 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 that 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 [3].

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 [4]. For at least two decades, HPC programmers have taken 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 and bandwidth of pins on a single chip means 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 peta-scale 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 [5].

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 shows 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.

The work that we currently pursue is the *initial phase* of a larger project in *Parallel Linear Algebra for Scalable Multi-Core Architectures*(*PLASMA*) that aims to address this critical and highly disruptive situation. While PLASMA's ultimate goal is to create software frameworks that enable programmers to simplify the process of developing applications that can achieve both high performance and portability across a range of new architectures, the current high levels of disorder and uncertainty in the field processor design make it premature to attack this goal directly. More experimentation is needed with these new designs in order to see how prior techniques can be made useful by recombination or creative application and to discover what novel approaches can be developed into making our programming models sufficiently flexible and adaptive for the new regime.

Preliminary work we have already done on available multi-core and heterogeneous systems, such as the IBM CELL processor, shows that techniques for increasing parallelism and exploiting heterogeneity can dramatically accelerate application performance on these types of systems. Other researchers have already begun to utilize these results. Under this early PLASMA project, we are leveraging our initial work in the following three-pronged research effort:

- *Experiment with techniques* – Building on the model of large grain data flow analysis, we are exploring techniques that exploit dynamic and adaptive out-of-order execution patterns on multi-core and heterogeneous systems. Early experiences with matrix factorization techniques have already led us to the idea of dynamic look-ahead, and our preliminary experiments show that this technique can yield great improvements in performance.
- *Develop prototypes* – We are testing the most promising techniques through highly optimized (though neither flexible nor portable and thus not general enough) implementations that we, and other researchers in the community, can use to study their limits and gain insight into potential problems. These prototypes are also enabling us to assess how well suited these approaches are to dynamic adaptation and automated tuning.
- *Provide a design draft for the PLASMA framework* – An initial design plan for PLASMA frameworks for multi-core and hybrid architectures is being developed and, in combination with PLASMA software prototypes, will be distributed for community feedback.

Though it is clear that the impact of the multi-core revolution will be ubiquitous, we believe that, in developing a programming model for this radically different environment, there are clear advantages to focusing on Linear Algebra (LA) in general and Dense Linear Algebra (DLA) in particular, as PLASMA does. For one thing, DLA libraries are critically important to Computational Science across an enormous spectrum of disciplines and applications, so a programming framework of the type we envision for PLASMA will certainly be indispensable and needs to be achieved as quickly as possible. But DLA also has strategic advantages as a research vehicle, because the methods and algorithms that underlie it have been so thoroughly studied and are so well understood. This background understanding will allow us to devise techniques that maximally exploit the resources of the microprocessor platforms under study.

As a third point, we claim that the techniques developed for LA are general enough to be utilized in other software libraries. In this respect, the research performed on the PLASMA project is expected to be beneficial for other libraries. Historically this has been the case with LAPACK and its use of the BLAS. Nowadays several libraries outside the LA discipline have followed the LAPACK example and their performance heavily relies on the BLAS. The inverse is not true, and we can not find such a generality in other disciplines.

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 proportionally increase*; 2) *the number and bandwidth of pins on CPUs are reaching their limits* and 3) *there will be a strong drift toward hybrid installations 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 can not have a sustained increase, 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, cornerstone numerical libraries obsolete, and how innovative techniques can exploit their parallelism and heterogeneity to address these problems.

2.2 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 comprehensive and widely used. The LAPACK/ScaLAPACK libraries 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 is increased while the problem size per processor is kept constant, effectively increasing the overall problem size. This measure tells us when we can exploit parallelism to solve larger problems. In this approach, increasing 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. Many problems in scientific computing reach their scaling limits on a certain number of processors determined by the ratio of computation/communication. With the speed of individual cores in the system on a decline, those problems will require *increased time to solution on the next generation of architectures*. In order to address the issue, emphasis has to be shifted from weak to *strong scaling*, where speed is achieved when the number of processors is increased while *the overall problem size is kept constant*, which effectively decreases the problem size per processor. In other words we need to seek more parallelism in algorithms and push their existing scaling limitations by investigating parallelization at a much finer levels of granularity.

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*.

2.3 Preliminary Work: Exploiting Parallelism on Multi-core

We used the forgoing analysis of the problems of LAPACK/ScaLAPACK on multi-core systems as the basis of some preliminary tests of techniques for doing fast and efficient LA on multi-core. LA operations are usually performed as a sequence of smaller tasks; it is possible to represent the execution flow of an operation as a Directed Acyclic Graph (DAG) where the nodes represent the sub-tasks and the edges represent the dependencies among them. Whatever the execution order of the sub-tasks is, the result will be correct as long as these dependencies are not violated. This concept has been used in the past to define "look-ahead" techniques that have been extensively applied to the LU factorization . Such methods can be used to remedy the problem of synchronizations introduced by non-parallelizable tasks by overlapping their execution with the execution of more efficient ones [1]. Although the traditional technique of look-ahead usually provides only a static definition of the execution flow that is hardwired in the source code, the idea of out-of-order execution it embodies can be extended to broader range of cases, where the execution flow is determined at run time in a fully dynamic fashion. With this dynamic approach, the subtasks that contribute to the result of the operation can be scheduled dynamically depending on the availability of resources and on the constraints defined by the dependencies among them (i.e., edges in the DAG).

Our recent work shows how the one-sided factorizations, LU, QR and Cholesky can benefit from the application of this technique [2]. Block formulations of these three factorizations, as well as many other one-sided transformations, follow a common scheme. In a single step of each algorithm, first operations are applied to a single block of rows or columns, referred to as the panel, then the result is applied to the remaining portion of the matrix. The panel operations are usually implemented with Level 1 and 2 BLAS and, in most cases, achieve the best performance when executed on a single processor or a small subset of all the processors used for the factorization.

It is well known that matrix factorizations have left-looking and right-looking formulations. The transition between the two can be done by automatic code transformations, although this requires more powerful methods than simple dependency analysis. In particular, the technique of look-ahead can be used to significantly improve the performance of matrix factorizations by performing panel factorizations in parallel with the update to the remaining submatrix from a previous step of the algorithm. The look-ahead can be of arbitrary depth, as was shown, for example, in the High Performance LINPACK benchmark (HPL). The look-ahead simply alters the order of operations in the factorization. A great number of permutations is legal, as long as algorithmic dependencies are not vi-

```
while(1)
   fetch_task();
   switch(task.type) {
      case PANEL:            // reduce the panel
         dgetf2();
         update_progress();
      case COLUMN:           // update a block-column
         dlaswp();           // of the trailing submatrix
         dtrsm();
         dgemm();
         update_progress();
      case END:              // perform left swap and return
         for()
            dlaswp();
         return;
   }
}
```

Fig. 1. Pseudo-code showing the execution flow for the LU factorization. The same execution scheme applies to the other one-sided transformations Cholesky and QR.

olated. From this point of view, right-looking and left-looking formulations of a matrix factorization are on two opposite ends of a wide spectrum of possible execution paths, with the look-ahead providing a transition between them. If the straight right-looking formulation is regarded as one with the look-ahead of zero, then the left-looking formulation is equivalent to the right looking formulation with the maximum possible look-ahead for a given problem.

Applying the idea of dynamic execution flow definition to LU factorization leads to the implementation of the left-looking variant of the algorithm, where the panel factorizations are performed as soon as possible, with the modification that if the panel factorization introduces a stall, then an update to a block of columns (or rows) of the right submatrix is performed instead. The updating continues only until next panel factorization is possible. Figure 1 (above) shows the simplified code that defines the execution flow. Here the steps of checking dependencies and making a transition are merged into the step of fetching the next task (the `fetch_task()` subroutine), where the choice of transition is made dynamically at run-time depending on the progress of the execution.

Experimental results show how the dynamic workflow technique is capable of improving the overall performance while providing an extremely high level of portability. Figure 2 shows that by applying dynamic task scheduling to the QR factorization, it is possible to out perform a standard LAPACK implementation with threaded BLAS.

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

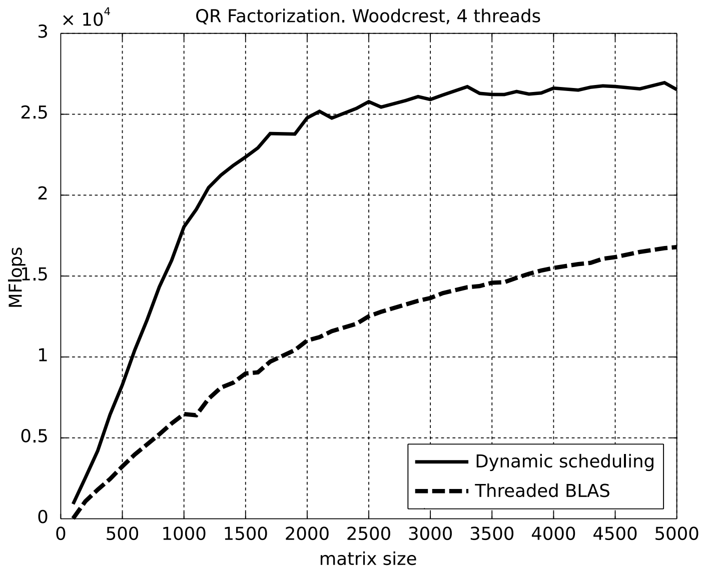

Fig. 2. Comparison of parallelization techniques for QR factorization (Two dual core Intel 3.0GHz Woodcrest processors (four cores total), GOTO BLAS 1.05, blocksize NB=64)

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.

References

1. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: Past, Present, and Future. Concurrency and Computation: Practice and Experience 15(9), 803–820 (2003), `http://www.netlib.org/benchmark/hpl/`
2. Kurzak, J., Dongarra, J.J.: Implementation of Linear Algebra Routines with Lookahead - LU, Cholesky, QR. In: Workshop on State-of-the-Art in Scientific and Parallel Computing, June, 2006, Umea, Sweden (2006)
3. Post, D.E., Votta, L.G.: Computational Science Demands a New Paradigm. Physics Today 58(1), 35–41 (2005)
4. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 30(3) (March 2005)
5. Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley, Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183 (December 18, 2006), `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`