SAND2008-3944C

Design Issues for Numerical Libraries on Scalable Multicore Architectures

Michael A. Heroux

Distinguished Member of the Technical Staff, Sandia National Laboratories, PO Box 5800 Albuquerque, NM 87185-1320

E-mail: maherou@sandia.gov

Abstract. Future generations of scalable computers will rely on multicore nodes for a significant portion of overall system performance. Presently most applications and libraries cannot exploit multiple cores beyond running addition MPI processes per node. In this paper we discuss important multicore architecture issues, programming models, algorithms requirements and software design related to effective use of scalable multicore computers. In particular, we focus on important issues for library research and development, making recommendations for how to effectively develop libraries for future scalable computer systems.

1. Introduction

Numerical libraries for parallel computers have traditionally had two separate target machine architectures: (i) highly scalable distributed memory systems with serial nodes and (ii) shared memory systems with modest scalability. Although shared memory nodes have been part of large-scale systems for many years, most scalable applications have not explicitly used shared memory programming on the node, instead relying on the message passing library, MPI in particular, to exploit shared memory for message passing performance.

The next generations of high end computer systems will combine scalable node count with shared memory nodes in ways that will likely require most applications and libraries to explicitly managed shared memory resources in addition to performing message passing across nodes, thereby requiring libraries to address both types of parallel resources. In this paper we discuss the architectural features libraries must exploit and support, and the implications for library developers and users.

2. Important Trends in Computer Architecture

For more than 15 years scalable computing has been dominated by single-level message-passing applications and computer systems. These are applications that view the computer system as a flat network of equi-distant, homogeneous processors, and computer systems that are able to behave as such, even though very few were physically configured that way. For a brief period of time around the year 2000, the parallel computing community thought that systems composed of networked shared memory parallel (SMP) nodes could not be viewed as a flat network and still allow optimal performance [1]. There was a general impression that such systems would require explicit shared memory programming underneath message passing in order to achieve best performance, e.g., MPI across nodes and OpenMP within a node. However, once message

passing libraries were made "SMP-aware" (such that messages between processors on the same node bypassed the network, thereby reducing network performance bottlenecks) the flat message-passing-only approach (where on a machine with m nodes and k cores on each node, we run an MPI program with $p = m * k$ processes in the traditional single-program-multiple-data (SPMD) model) continued to be the best overall approach to scalable application design, and continues to be true even today on most systems.

Despite the success of the single-level, message-passing-only approach, there is growing evidence that future applications and libraries will need to address increasing node complexity by introducing another level of parallelism. There is little doubt that message passing will still be the dominant inter-node programming model, but there is also much doubt that it will be sufficient for driving all cores on a single node for much longer, at least at a single-level. As a result, we must explore new node programming models that can be effectively used in combination with message passing across nodes.

There are many ways to characterize multicore architectures, but for our purposes we consider two classes: homogeneous and heterogeneous. For each class we discuss important hardware and software issues that are impacting library software design and implementation.

2.1. Homogeneous Multicore Nodes

Homogeneous multicore nodes, e.g., Intel and AMD quad-cores and Sun's Niagara2, have identical cores with shared memory (although memory access may be non-uniform). Because of their architectural predecessors, SMP nodes, these nodes have several well-established parallel programming models, such as Posix threads and OpenMP. Furthermore, as long there is sufficient memory bandwidth, message passing is often a very effective approach.

Homogeneous multicore nodes are in many ways similar to microprocessor SMP architectures of the past 15 years. The biggest differences are that multicore nodes are now available in every computer and core counts are expected to increase substantially. Previous multiprocessor architectures were primarily available only in servers, and two or four processors were the most common configurations.

2.1.1. Slow Growth in Single Core Performance For many years, users of commodity microprocessors could realize regular performance improvements in their applications simply due to single processor performance improvements. Many of these improvement were due to increased clock speeds and sophisticated instruction scheduling in hardware. Presently clock speed are staying relatively constant and core architectures are expected to become simpler, if change much at all. The effect is that single core performance is improving but not as rapidly as in the past, and users who want to once again experience substantial performance improvements from one generation of microprocessor design to the next must learn to exploit multiple cores.

Although not an exhaustive list of machines, Table 1 shows the performance profile over 5 years of processors for HPCCG, an unstructured finite element microapplication. HPCCG performance is a function of both floating point capabilities and memory system performance. Therefore single core performance rates will increase due to memory system improvements, even if floating point rates stagnate. However, if we restrict HPCCG execution to a single core, the vast majority of performance improvement will be missed since parallel execution across multiple cores enables an additional three times performance improvement on four cores (Figure 1 red line).

2.1.2. Weak per-core Memory Bandwidth Although single core performance growth is slowing, if we can exploit multiple cores, we will continue to see single node performance improve. However, in addition slow growth in single core performance, current multicore nodes do not tend to have sufficient memory bandwidth to adequately support all cores when performing

Table 1. Single core performance of HPCCG, an unstructured finite element microapplication, on several machines. 3D problem with 1 million mesh nodes. For each machine we list the year purchased, the processor description and the MFLOPS/s for the HPCCG execution.

Year	Processor	Speed (GHz)	Cores (per socket)	(MFLOPS/s)
2003	AMD Athlon	1.9	1	178
2004	AMD Opteron	1.6	1	282
2005	Intel Pentium M	2.1	1	310
2006	AMD Opteron	2.2	2	359
2007	Intel Woodcrest	1.9	4	401
2007	AMD Opteron	2.1	4	476
2007	Intel Core Duo	2.3	2	508

memory intensive computations. This implies that memory-bound algorithms will be unable to fully utilize all cores. Figure 1 shows the performance of pHPCCG, a parameterized version of HPCCG that supports arbitrary data types, on one to eight cores of an AMD Barcelona node. On single core scalable systems, HPCCG scales linearly to more than 10,000 cores. However, in this study, linear speedup was achieved only for single precision data going from one to two cores. All other data points, single or double precision, show sublinear improvement.

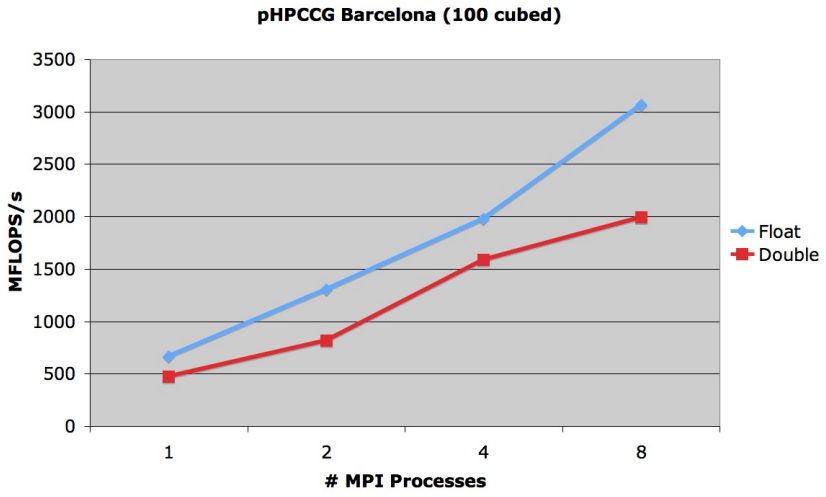

Figure 1. Comparison of single vs. double precision performance for pHPCCG, an unstructured implicit finite element microapplication. Results are for 1M equations per process using 1-8 MPI processes on a dual chip quadcore AMD Barcelona node.

2.1.3. Multithreading Multithreading is a well-known architectural approach to reducing effective latencies by keeping multiple thread states. If the active thread on a core stalls, the core can attach to another thread that is not stalled. This capability is especially effective for bandwidth-intensive programs and those with irregular memory access patterns. A processor's ability to have multiple outstanding threads can greatly improve performance, often without sophisticated software restructuring, which is otherwise typically required on more traditional microprocessors. The Sun Niagara 2 processor contains eight cores with eight threads per core. Figure 2 shows performance for Trilinos Epetra kernels [2], and that oversubscribing cores with MPI processes can lead to substantial performance improvements. Multithreading will likely be a part of other new node architectures and exposing a large collection of parallel task can greatly improve core utilization.

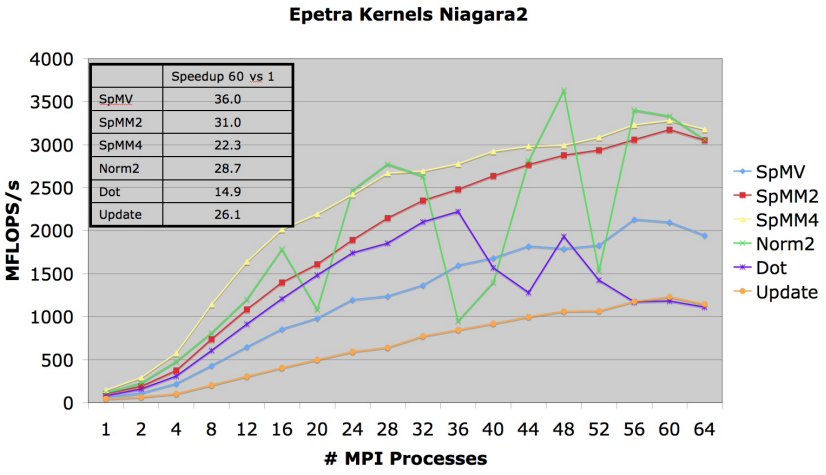

Figure 2. Weak scaling study of Epetra using 1-64 MPI processes on Sun Niagara2. Although there are only eight cores, performance improves due to multithreading until about 60 processes. Dot product and two-norm performance peak at a speedup of 28.0 and 31.3 using 36 and 48 processes, respectively.

2.1.4. Core Count Growth When SMP nodes became available about 15 years ago, there was some expectation that processor counts per node would steadily increase on desktop systems. This expectation was not fully realized. Presently, we have a similar expectation that core counts on homogeneous multicore systems will steadily increase. If true, we will see tens of cores per node and programming these cores using a single-level flat MPI model will be difficult. However, there are a number of implementation details that can improve MPI performance for large core counts, which we discuss in Sections 3.1.1 and 3.1.2.

2.2. Heterogeneous Multicore Nodes

Heterogeneous nodes, e.g., STI Cell, nVidia and ATI GPUs, have two different core types (more in some cases). One processor is used primarily to handle serial and administrative tasks. The other cores are all of of the same type, are very capable of performing floating point computations, have a communications path to the serial processor and also have their own high performance memory network. Most heterogeneous multicore nodes have emerged from the multimedia computing field and each has its own unique programming environment. Although efforts are underway to define a common programming environment for these nodes, presently there is no way to easily write portable code for these nodes.

2.2.1. Native Double-precision Support Heterogeneous multicore nodes have been of interest to the high performance computing community for many years [3]. However, full support for double precision floating point arithmetic on these systems has only recently become available. As a result, prior to now, very few scientific and engineering applications could use heterogeneous multicore nodes in a production setting. Some floating point behavior differences still exist, such as lack of support for gradual underflow, but in most situations the degree of double precision support is sufficient for applications.

2.2.2. Programming Interfaces Perhaps the biggest hurdle to using heterogeneous multicore nodes is the lack of a standard, portable programming interface. Each node vendor has its own interface, and some commercial products are available to provide a generic interface, but presently there is no open standard programming interface that works across multiple heterogeneous multicore nodes. Therefore, in order to use these nodes, we must write our software in the native environment of each node, or provide our own abstract interface with adapters for each node type.

3. Node Programming Models

Since message passing, specifically MPI, will be the dominant programming model for parallel execution across multicore nodes, the real issue for libraries on future scalable systems is how to program the node. The simplest approach is to use a single-level of MPI processes, and there is some consensus that single-level MPI-only will be a sufficient programming model for most scalable systems for about three to five more years, unless of course an application must be ported to systems such as the Los Alamos Roadrunner [4], which rely on STI Cell processors.

Beyond the three to five year time frame, there is no general consensus on how to program multicore nodes. We will likely have many programming models. This issue is probably the single most challenging for library designers, since library software must be embedded into many different software frameworks and cannot generally control its runtime environment.

3.1. MPI under MPI

Although one might assume that a shared memory multicore node should use a shared memory programming model, there is merit to considering MPI underneath MPI. In this case, we not only use MPI across nodes, but also program each multicore node via a local MPI communicator. This approach is attractive because an application and its developers rely upon only on a single programming model.

Furthermore, the interoperability of MPI with other parallel programming environments can be an issue at runtime. Process scheduling and other runtime resource management problems can lead to difficult-to-debug performance degradation. Even efficient MPI-only execution can be difficult on multicore systems and we will need to revisit MPI implementations.

3.1.1. Multicore-aware MPI Similar to making MPI SMP-aware 15 years ago, MPI implementations must now become "multicore-aware." In addition to performing shared memory copies when communicating between shared memory cores, MPI runtime environments should keep a single copy of the application executable per shared memory image, especially as core counts increase. This optimization is already available on some systems, e.g., Sandia's Redstorm.

MPI process placement and memory allocation are also an issue. Figure 3 shows the before and after performance impact of poor memory placement policy. Since the AMD Barcelona node has a non-uniform memory access (NUMA) design but is logically shared, memory can be allocated non-optimally for an MPI process. Present runtime environments do not take into account the importance of memory and process affinity. Although there are often system tuning functions that an experienced programmer can use to address these issues, a multicore-aware MPI environment should handle these problems automatically.

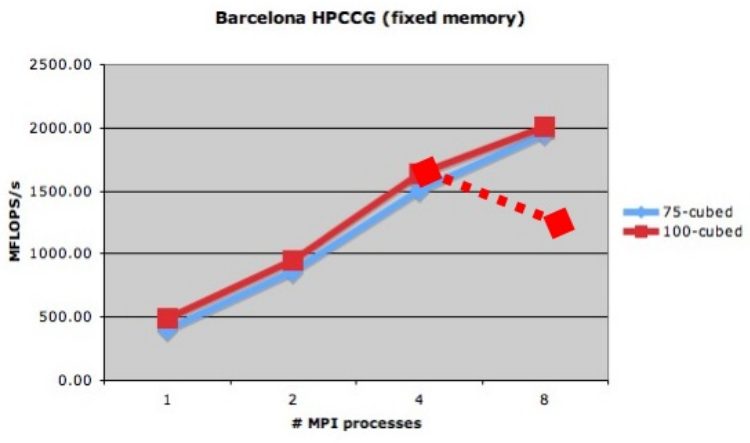

Figure 3. Comparison of performance for HPCCG, an unstructured implicit finite element microapplication, before and after memory placement policy changes.

3.1.2. Shared-memory MPI Beyond fixing MPI runtime performance so that standard subcommunicators can be used efficiently on multicore nodes, some modest additions to MPI can make shared memory parallel programming feasible and portable. Specifically, by supporting a shared memory allocation such that one MPI process can allocate memory that is also visible to other processes on a node, we can implement parallel, fine-grain shared memory algorithms using MPI. Using MPI barriers to control loop increments and ensure data integrity, MPI can easily be used to implement shared memory algorithms. Such mechanisms also set us up to consider hybrid runtime environments, which we discuss in Section 3.3

3.2. Threads under MPI

Perhaps the most natural way to program homogeneous multicore nodes is to use one of several thread-based programming models. Pthreads and OpenMP are the most prevalent threading models, developed 15-20 years ago for SMP nodes. Threads under MPI is already well-established as a two-level parallel model, used successfully by many applications to extract additional parallelism from their application, e.g., shared memory parallel FFTs within an MPI weather modeling application.

More recently, other threading models have emerged which provide richer runtime semantics that can improve work partitioning and scheduling, and integrate into advanced language environments. Perhaps most notable is the Intel Thread Building Blocks library (TBB) [5], which uses advanced features of C++ to support default and user-defined work partitioning, as well as dynamic thread scheduling. TBB is open source and extensible. It also has the potential to be useful on heterogeneous multicore nodes.

3.2.1. Thread Management Strategies and Multiple Thread Runtime Environments One of the more challenging aspects of thread programming is managing, placing and scheduling threads. Pthreads, OpenMP and similar models provide almost no control over these thread policies, or give control only at a global level via environment variables or system function calls. To make the situation even more challenging, there is no protocol for how these thread models should interact with each other within the same application. This is a particular problem for libraries that may use one model, e.g., OpenMP, and are called from an application that uses another model. Runtime performance issues are unpredictable and there are very limited, operating system specific mechanisms to handle thread policies.

Portable performance of thread-based parallelism is probably the single biggest challenge to using threads underneath MPI. When coupled with the fact that there is no portable programming model for heterogeneous multicore nodes, we come to realize that the single biggest barrier to shared memory parallel multicore programming is the lack of a programming model.

3.3. Hybrid Runtime Environments

A common observation in porting applications to homogeneous multicore-based scalable systems is that the vast majority of computations perform very well using single-level MPI-only as the programming model. In fact, for some applications that use third-party solver libraries, all of the application's computation can scale using MPI-only because it is only the solvers that are not scaling well. The solvers are not poorly designed or implemented. They are simply memory bandwidth bound, while the application has good data locality. Figure 4 shows the breakdown of time spent in the Fluid Density Functional Theories program called Tramonto. The setup time is the time spent in Tramonto itself, while the solve time is spent in library kernels. Although Tramontos profile is dramatic, other applications show similar behavior.

Based on this observation, developing data classes that can be used by an application in MPI-only mode such that the data is stored in a way that is accessible to a library via shared memory addressing on a multicore node becomes attractive. In this way the library can use

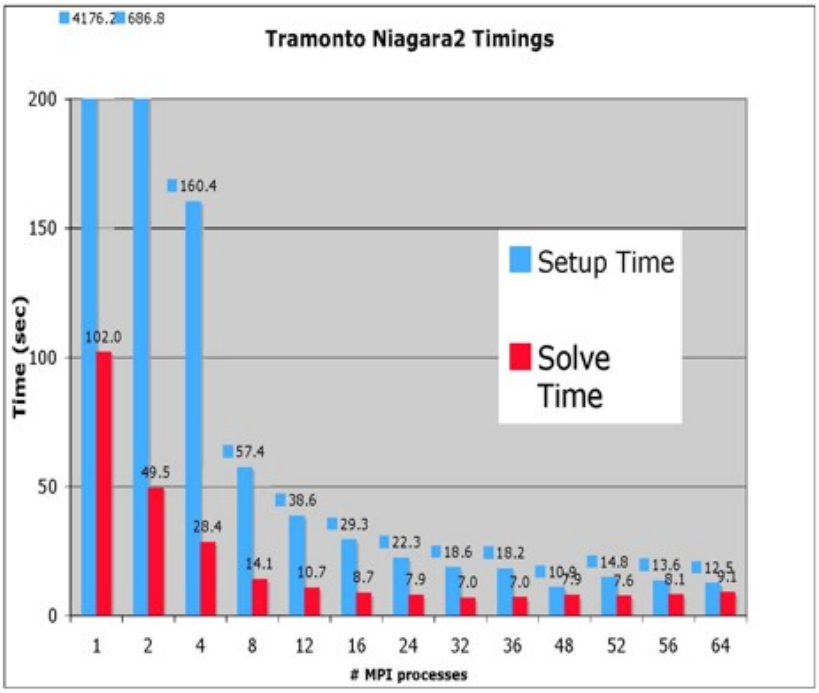

Figure 4. Tramonto is a classical density functional theories code for inhomogeous fluid modeling. Timing results are for 1-64 MPI processes on an 8-core, 8-threads-per-core Niagara2 processor. Setup Time is time spent in Tramonto except for the solver. Solve Time is spent in a Trilinos-based custom preconditioned solver. Note that outside of the solver, Tramonto scales extremely well using MPI-only.

shared memory algorithms to improve runtime performance and robustness without requiring any change to the application that is using the library. Although applications will need to become "multicore aware" at some point, by using a hybrid runtime environment we can significantly extend the years during which existing scalable applications can continue using single-level MPI only.

4. Designing Libraries for Scalable Multicore Architectures

High performance libraries have been an important part of scientific and engineering computing for decades, and their importance grows as machine architectures become more complex, and as libraries become better designed to integrate easily within an application. Despite the fact that there are many different science and engineering applications, the underlying algorithms used by these applications have remarkable similarities, especially those algorithms that are most challenging to implement well in parallel. It is not too strong a statement to say that libraries are essential to the broad success of scalable high performance computing [6].

Based on the above discussions about multicore trends and programming models, we now discuss what library developers can do to design and develop effective libraries for future scalable systems, as well as the challenges we face going forward.

4.1. Addressing Bandwidth Challenges

Common to all multicore architectures is bandwidth limited computations for sparse linear algebra and dense vector kernels. Because many applications use these kernels, we must look at algorithms and data structures that can reduce our bandwidth requirements.

4.1.1. Block Algorithms Block iterative methods (in contrast to single vector methods) for solving linear systems and eigenvalue problems have attractive performance behavior. Vector updates and dot products become (oddly shaped) dense matrix-matrix multiplies. Sparse matrix vector multiplication (SpMV) becomes multiplication by multiple vectors (SpMM).

Figure 2 shows the performance advantage that SpMM has over SpMV. Block methods are also attractive numerically because of robustness properties. Also, as we move toward mathematical optimization, uncertainty quantification and related higher level algorithms, solution of multiple simultaneous systems becomes a basic kernel. As a result, one of the most important ways to address reduce per-core bandwidth is to develop libraries of block iterative methods.

Similarly, blocking of sparse matrix data structures can be very effective. Although automated techniques such as those implemented in OSKI [7] have been shown to be effective for fast SpMV execution, overall efficiency is still best served by problem-defined blocking if it is present in the problem being solved. This is mentioned further in Section 4.3.

4.1.2. Multi-precision Data Types and Mixed-precision Algorithms Use of single precision (32-bit) arithmetic has always had some performance and storage advantage over double (64-bit) precision. However, when double precision became widely supported in hardware, most science and engineering applications, with some notable exceptions, moved to using double precision for all computations. The motivation for using double precision in some phases of computation is clear: robust results. However, the motivations for solely using double precision are not well documented in the literature, although are likely due to simplicity in code implementation and maintenance. As observed in Sections 2.1.1 and 2.1.2, there is growing incentive for judicious use of single precision data within double precision applications.

Mixed precision algorithms are well known in the numerical analysis literature. Although the most prevalent of these is iterative refinement [8], mixed-precision approaches are useful in many other settings. For example, using lower precision for grid transfer operators in multigrid and local element stiffness computations in finite element stiffness matrix calculations can be effective.

The first limitation to developing lower and mixed-precision algorithms is the limited ability to express multi-precision and precision-neutral algorithms in our most common computing languages. Although Fortran 2003 provides some support for generic programming, the only practical approach to implementing multi-precision and precision-neutral algorithms at this point is using templates in C++. In order to provide support for multi-precision algorithms, we need the following:

(i) Generic data classes whose precision can be declared when an instance of the class is created.
(ii) The ability to express an algorithm independent of precision.
(iii) A rich generic attribute set including the concept of zero and one, and a relative half or double precision.

Many existing scalable libraries support 32-bit or 64-bit data types, but often the choice must be made at compile time using the same namespace. Few libraries support the generic expression of an algorithm independent of the floating point precision and very few if any support the generic expression of a mixed precision algorithms. Using C++ templates and traits mechanisms, we can provide data classes that support multiple precisions and provide the ability to express generic and multi-precision algorithms.

4.2. Shared Memory Algorithms

If we are to benefit from shared memory multicore nodes beyond making MPI implementations multicore-aware, we will need to develop shared memory algorithms that are faster, more robust or both when compared to default single-level MPI-only execution. There are two primary ways to accomplish this:

(i) Exploit fine-grain data parallel algorithms.
(ii) Partition work and data for dynamic load balancing.

SPMD is not an effective programming model for algorithms that are highly parallel with fine-grain parallel data dependencies, or for task pools of work that are difficult to partition across processors *a priori*. For example, via level scheduling algorithms, sparse triangular solves have sufficient parallelism to keep multiple processors active, but the parallel tasks depend on shared memory access to data that was generated at a previous level. Furthermore, at each level, the parallel tasks vary greatly in the amount of work per task, so static scheduling is typically imbalanced. Figure 5 illustrates level scheduling and shows typical statistics for an unstructured problem. By using shared memory implementations of sparse triangular solves as a preconditioner or smoother for multi-level preconditioners we can reduce the number of subdomains in a domain decomposition preconditioner thereby reducing the scalability complexity of a parallel solver.

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 & 0 \\ l_{31} & 0 & 1 & 0 & 0 \\ 0 & 0 & l_{43} & 1 & 0 \\ l_{51} & 0 & l_{53} & 0 & 1 \end{bmatrix} \quad \text{Solve } Ly = x.$$

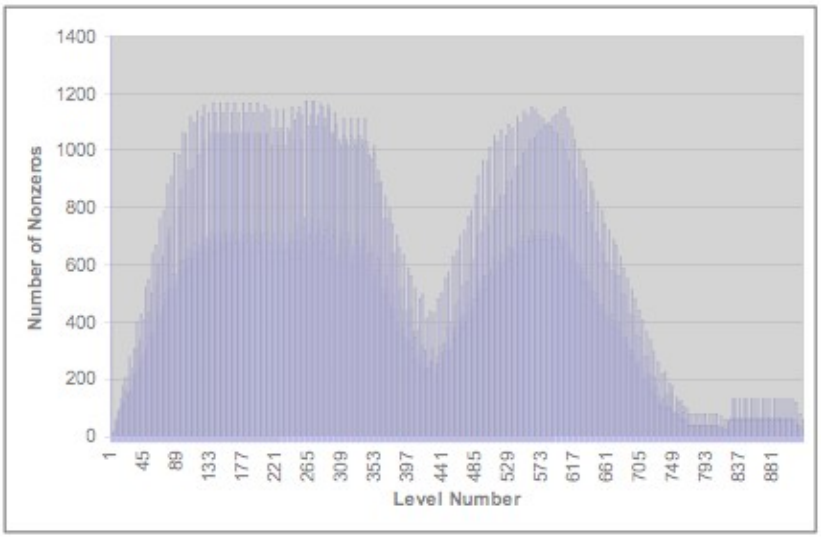

Figure 5. (a) Level Scheduling of lower triangular solve: After Row 1 is processed, Rows 2 and 3 can be processed in parallel, followed by Rows 4 and 5 in parallel. (b) Level statistics for a 3D unstructured finite element matrix.

4.3. Opaque Data Structures

As mentioned in Section 4.1.1, OSKI and other inspector-executor approaches have become popular for providing opaque data structures that provide a machine-friendly layout of data while hiding the details from the user. This approach has been used very successfully for level-3 dense BLAS in libraries such as ATLAS [9] where data transformations can occur as part of executing the kernel. It is less clear that similar approaches will work in the long run for other library kernels where the computation costs cannot cover the data transformations in real time and must be done as part of a preprocessing step. Although kernel execution time can be faster, production users are reluctant to adopt this degree of opaqueness for a variety of reasons including: duplication of data (the application's copy and the library's), *ad hoc* semantics of updating data, and amortization of the inspector phase costs. Opaque data structures will be necessary for optimal multicore execution. However, we have only started to design high-quality software that can be readily adopted by production application users.

4.4. Node API Abstractions

Multicore nodes are evolving rapidly. Although we have mature shared memory programming models for homogeneous multicore nodes, the runtime environments of these models are often poorly defined and difficult to control within the context of a scalable high performance application. Furthermore, heterogeneous multicore nodes appear to offer better long-term performance and presently have no portable open standard programming model. Because of

these facts, the most important software issue facing scalable libraries at this time is how to develop and define an abstract node programming interface that is compatible with and implementable on current platforms while extensible to future multicore platforms.

An effective approach to solving this problem is to use C++ abstract classes to define a parallel machine model. By describing the parallel machine in terms of attributes and services needed by the library, we have a chance to achieve efficient performance on current architectures and still retain flexibility to adapt to future ones. Presently no other reasonable approach exists.

5. Conclusions

Multicore nodes present an exciting and challenging environment for scalable application and library developers. For the first time in 15 years we face the real prospect of redesigning much of our software base. Libraries will play a critical role in several ways. They will provide some of the first multicore-optimized software that will enable scalable application performance and portability, and extend the period of time where single-level MPI-only programming still works within the application. Libraries will also provide prototype portable programming interfaces for multicore nodes that can serve as candidates for open standards.

As we move forward, library developers must focus efforts on reducing memory bandwidth requirements by use of new algorithms, lower precision arithmetic and improved data structures. We must also focus on shared memory algorithms beyond a multicore-aware MPI, enabling spawning of multiple tasks that can be dynamically scheduled. Finally we must develop abstraction layers that support efficient opaque data structures and robust node programming interfaces that allow us to adapt as multicore architectures and programming models evolve.

As a final note, it is clear from the above discussion that advanced object-oriented programming, using C++ as the primary language, is the most effective approach to next-generation library development. Although Fortran 2003 (and soon 2008) contains many attractive features, access to a standard-compliant Fortran 2003 compiler is limited. Furthermore, C++ offers the best combination of access to robust compilers, language interoperability, true object-oriented support and compile-time polymorphism (via templates) for high-performance generic data types and mixed precision algorithms. Adopting C++ does not mean that Fortran should be abandoned, but it does mean that Fortran applications will need to link with C++ easily and portably.

References

[1] Krawezik G 2003 *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA: ACM) pp 118–127 ISBN 1-58113-661-7

[2] Heroux M A, Bartlett R A, Howle V E, Hoekstra R J, Hu J J, Kolda T G, Lehoucq R B, Long K R, Pawlowski R P, Phipps E T, Salinger A G, Thornquist H K, Tuminaro R S, Willenbring J M, Williams A and Stanley K S 2005 *ACM Trans. Math. Softw.* **31** 397–423 ISSN 0098-3500

[3] See homepage for details General-purpose graphics processing units (gpgpu) homepage http://www.gpgpu.org/

[4] See homepage for details Los alamos roadrunner homepage http://www.lanl.gov/roadrunner/

[5] Pheatt C 2008 *J. Comput. Small Coll.* **23** 298–298 ISSN 1937-4771

[6] Kothe D and Kendall R 2007 Computational science requirements for leadership computing Tech. Rep. ORNL/TM-2007/44 Oak Ridge National Laboratory

[7] Vuduc R, Demmel J W and Yelick K A 2005 *Journal of Physics: Conference Series* **16** 521–530 URL `http://stacks.iop.org/1742-6596/16/521`

[8] Buttari A, Dongarra J, Langou J, Langou J, Luszczek P and Kurzak J 2007 *Int. J. High Perform. Comput. Appl.* **21** 457–466 ISSN 1094-3420

[9] See homepage for details Atlas homepage http://math-atlas.sourceforge.net/