Some Essential Techniques for Developing Efficient Petascale Applications

Laxmikant V. Kalé

Parallel Programming Laboratory, Computer Science Department, University of Illinois at Urbana-Champaign

E-mail: kale@cs.uiuc.edu

Abstract. Multiple PetaFLOPS class machines will appear during the coming year, and many multi-PetaFLOPS machines are on the anvil. It will be a substantial challenge to make existing parallel CSE applications run efficiently on them, and even more challenging to design new applications that can effectively leverage the large computational power of these machines. Multicore chips and SMP nodes are becoming popular and pose challenges of their own. Further, a new set of challenges in productivity arise, especially if we wish to have a broader set of applications and people to use these machines. I will review a set of techniques that have proved useful in my work on multiple parallel applications that have scaled to tens of thousands of processors, on machines like Blue Gene/L, Blue Gene/P, Cray XT3 and XT4. I will identify new challenges and potential solutions for the performance issues. Issues presented by multicore chips and SMP nodes will also be addressed. Finally, I will review some new and old ideas for increasing productivity in parallel programming substantially.

1. Introduction

As I write this paper, the first petascale supercomputer has already been assembled. Many supercomputers with multiple petaFLOPS peak performance are being developed to be deployed within the next few years. Which parallel programming techniques are necessary to harness this unprecedented power for science and engineering applications that could lead to breakthrough discoveries? In this paper, I will describe my answer to this question, based on my research group's experience with developing multiple parallel applications that have scaled to tens of thousands of processors.

Multiple trends are contributing to the emergence of powerful parallel machines. The first is the long-standing trend of growth in computer hardware, including processors and memories getting cheaper. Of course, processors have stopped becoming faster in terms of clock frequency, but that is after several decades of continuous improvement on that count as well. As a result, it has been feasible to build very large parallel computers with relatively modest investments. For example, a company in India was able to build a computer with a peak performance of 170 teraFLOPS [27], reportedly for a cost of about 30 million dollars.

The second dramatic trend is that of an increasing number of cores on each multicore chip. Clock frequencies cannot be increased any further due to power dissipation concerns. Yet the feature size keeps decreasing and the number of transistors per chip keeps going up. Hence, it makes sense to organize all those transistors into multiple processor cores on a chip. Intel's eight-core chip is expected to ship later in 2008; further it is expected that the number of cores

per chip will keep doubling every so often (say 18 months) for at least a decade. The ways in which the mainstream desktop market responds to these chips will have significant consequences for the high-performance computing community.

The third trend is the development and incorporation of specialized accelerator chips and accelerator-like features in mainstream chips. Accelerators, by restricting their domain of application to a few specific patterns of computation, have been able to achieve performance that is orders of magnitude higher than the general purpose processors. Graphics processors have an extremely specialized function, of computing pixels, given a set of polygons. Yet, they were used by enterprising computational scientists for carrying out computations useful for their applications. But now these accelerators have evolved towards a somewhat more general architecture, as seen by NVIDIA's new architecture and the CUDA programming system. IBM's Cell Broadband Engine presents an architecture that is more general purpose than GPGPUs, but still specialized compared with mainstream processors. Each of the eight specialized cores on the chip must work with only a small scratch-pad memory, with explicit DMA for data movements between the scratch-pad and main memory.

At the same time, there exist computational modeling problems which can be solved if one can harness this compute power effectively, which will lead to dramatic benefits for mankind. There are breakthroughs that are waiting to happen in multiple areas. It can improve our understanding of nature: biophysical simulations for understanding the molecular machinery of our bodies, genetic computations to understand evolution, similarities and differences between individuals, and the relationships between the genotype and phenotype. It can improve our ability to predict future: including short-term weather and long-term climate predictions and possibly even global economic predictions. It can lead to better design of engineered artifacts, such as automobile engines, rockets or buildings.

The new generation of CSE applications do not just exploit the larger compute power by "increasing resolution everywhere". Instead, they tend to use techniques such as adaptive and dynamic mesh refinements, sophisticated algorithms such as Barnes-Hut or Fast Multipole Method, and are therefore more challenging to parallelize effectively.

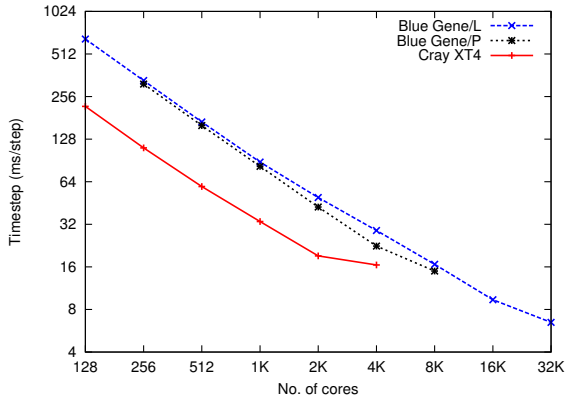
To summarize, parallel computers of the petascale era are here and they are likely to have a rich, complex and possibly heterogeneous structure. High-impact science and engineering applications that need such power exist or are being developed, but are challenging to parallelize effectively. The parallel programming challenge before us in the coming era is: how to productively and efficiently program the multi-petaFLOPS machines? In this paper, I will outline some of the lessons I have learned in parallelizing several applications to machines with over $40,000$ processors, that I believe will be useful to address this challenge.

1.1. Application Experience

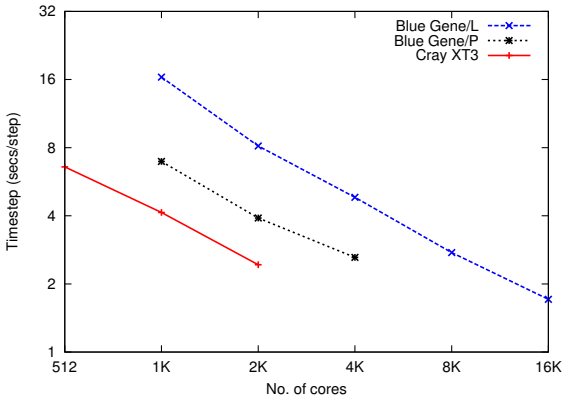
I will select examples from the experience of my research group[1] in parallelizing a few CSE applications that are used by CSE communities in their regular research: NAMD, from our oldest collaboration, is a program for biomolecular modeling [2]; OPENATOM is a Car-Parinello MD program used for simulation of electronic structure (in nanomaterials, as well as biophysics) [3]; ChaNGa, an astronomy code [13]; and RocStar, a code for simulating solid-propellant rockets, such as those in the space shuttle solid rocket booster. Figures 1(a), 1(b) and 1(c) show performance of these applications on three machines: the Blue Gene/L at IBM (up to $32,768$ processors), the recent Blue Gene/P at ANL, and the Cray XT3/4 at PSC and ORNL.

The success of these applications, which are used routinely by scientists, substantiates and leads credence to the lessons presented in the rest of the paper. However, I note that the SciDAC community, and the broader HPC community has a significant experience in developing highly

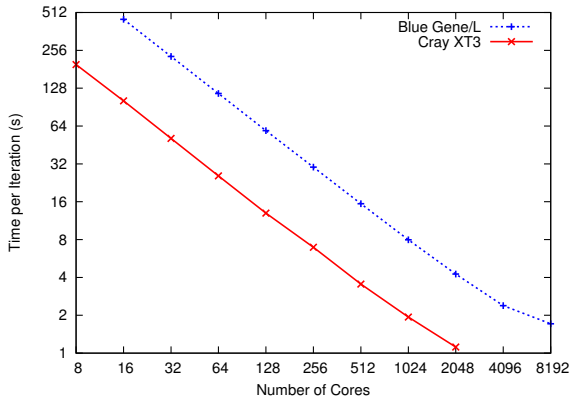
[1] Parallel Programming Lab, UIUC – http://charm.cs.uiuc.edu



(a) NAMD running STMV (1 million atoms)



(b) OPENATOM running WATER_256_70Ry (256 water molecules)



(c) ChaNGa running hwrh_lcdms (16 million particles)

Figure 1. Performance of NAMD, OPENATOM and ChaNGa on Cray XT3, XT4 and IBM Blue Gene/L, Blue Gene/P

scalable experience. So, the "lessons" or dicta presented here should be taken as a subjective selection based on my experiences and my set of techniques. A union of such lessons and techniques from multiple HPC researchers will be needed to tackle the challenge of petascale application development.

2. Guiding Principles

Our computer science research, towards the goals mentioned in the introduction, involves development of abstractions and techniques that help simplify parallel programming and improve performance of applications developed. We use the following guiding principles to decide which abstractions and techniques to explore.

- **Don't rely on magic (at least until you have learned the trick):** For example, compilers that automatically parallelize sequential codes is an attractive research idea, but one that smacks too much of wishful thinking. Parallelizing compilers have achieved close to technical perfection, in that they can extract most of the parallelism *that can be extracted* from sequential programs, with a huge intellectual effort by the compiler community. However, they have proved inadequate mostly because sequential programs obscure too much information. In general, it is better to develop abstractions in a bottom-up manner. One aims at developing an abstraction that builds on existing technology, and one that can

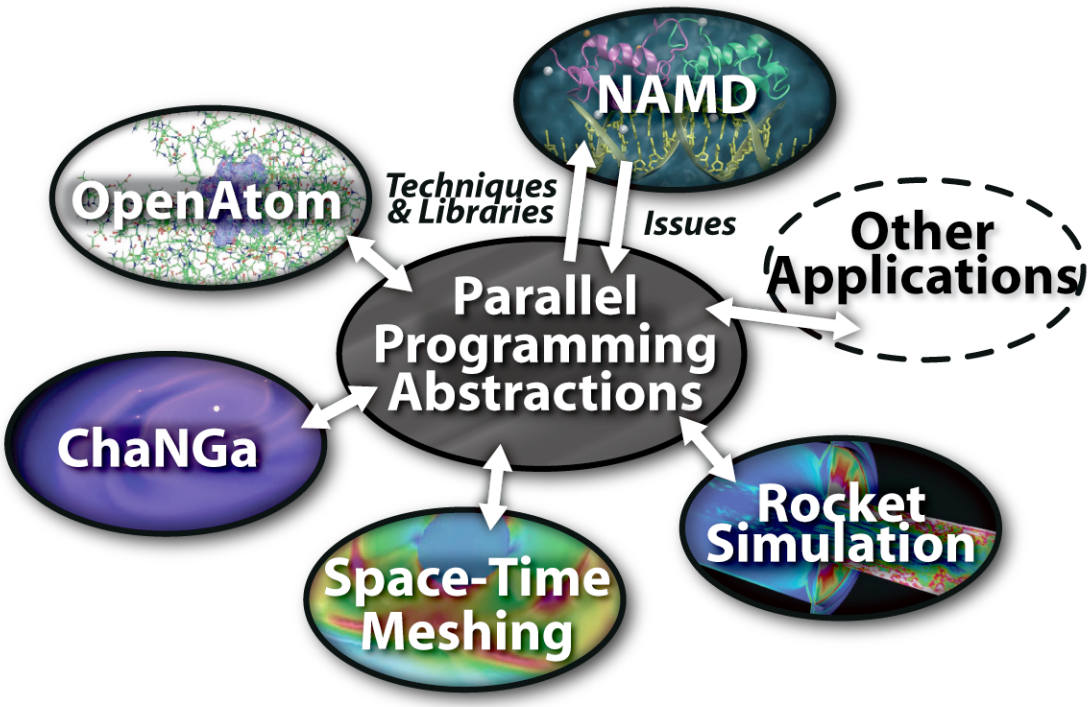


Figure 2. Application Oriented Parallel Abstractions

be seen to be feasible at the outset, and one that can be shown to be useful (in raising the level of abstraction) for realistic applications. Of course, such an organic bottom-up growth of abstractions needs to be controlled with the quest for elegance and simplicity.

- **Seek an optimal division of labor between the system and the programmer:** There are tasks that a human programmer can do much better than any automated system and vice versa. By automating those tasks that humans have most difficulty with, while avoiding automating tasks which are difficult to automate (such as automatic parallelization) but are not that difficult for an application programmer, we will get the most benefit from our research efforts.
- **Design abstractions based solidly on use cases:** As computer scientists, we tend to develop abstractions in an almost "platonic" way. Elegance and beauty may influence us more than relevance to real applications. This tendency may lead to abstractions that one can write papers about but that are not used by the CSE community, and thus fail to have an impact on the state of the art. At the other extreme, a computer scientist may help parallelize an individual application, but then the contributions are not transferable to other applications. Instead, one should develop abstractions in the context of multiple full-scale applications. Benchmarks, short algorithms or kernels don't expose the full complexity of the parallel programming problem, nor does a single application. If the abstractions that are triggered from individual applications are used, tested and honed in the context of multiple applications, they are likely to improve the state-of-art in parallel programming. Figure 2 illustrates this idea, in the context of my group's abstractions, and some of the applications developed with this paradigm. We called this approach "Application-oriented yet Computer Science centered research" [14].

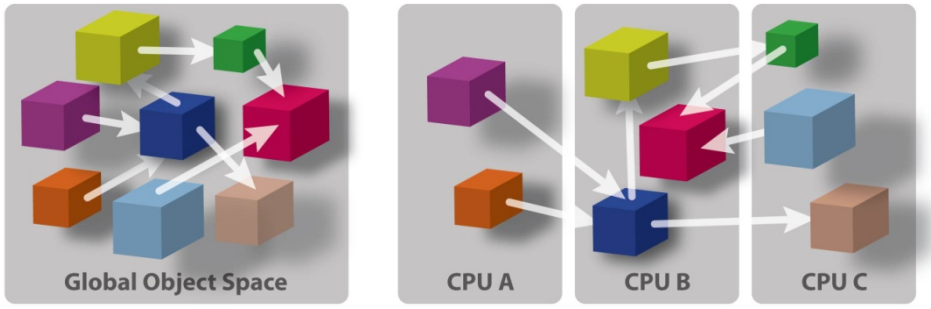


Figure 3. The user is only concerned with objects and communication between them. Actual mapping of objects to processors and underlying messaging is handled by the system.

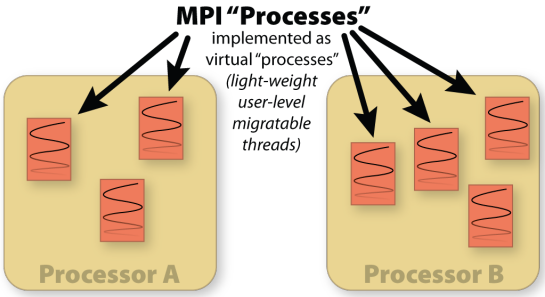


Figure 4. Virtualization for MPI Threads using Adaptive MPI

3. Lessons Learned

In the following subsections, I will enumerate a few "imperatives" of petascale programming. Each is based on a technique (or set of techniques) that has been proved useful in highly scalable CSE applications, and that I consider to be crucial to programming the next generation of parallel supercomputers.

3.1. Decouple Decomposition from Physical Processors

Dictum 1: *Use over-decomposition with migratable objects, and leverage its benefits among multiple dimensions via an intelligent runtime system.*

Processor virtualization based on migratable objects was developed in the early 1990s in my group's work on Charm++. The basic idea is simple: let the programmer over-decompose the computation and data into logical work-units and data-units, and let an intelligent runtime system (RTS) assign these units to physical processors. Several others have developed similar ideas since early days of parallel computing [7, 24]. The decomposition done by the programmer is independent of the number of processors. Yet, the programmer does exercise grain-size control, ensuring that the overhead is adequately amortized by useful computation in each grain. Since the overhead is independent of the number of processors, the programmer does not have to optimize for the particular number of processors available in their allocation, in most cases.

Charm++ and AMPI are two systems that embody this strategy. In Charm++, the data and the computation is decomposed into multiple indexed collections of data-driven migratable objects that communicate with each other via asynchronous method invocations, and a few asynchronous collective calls (Figure 3). AMPI is an implementation of MPI where each MPI "process" is implemented as a user-level light weight migratable thread (not to be confused

with a POSIX thread, see Figure 4). Over the past decade (and more), we have learned that this strategy yields a multitude of rich dividends. The first category of benefits is in the area of better software engineering, whereas the second category consists of benefits arising out of automated program optimizations via intelligent runtime strategies enabled by migratable objects. A longer, albeit somewhat dated, exposition of these benefits can be found in [16, 17]. Here, a brief summary is presented.

Separating the decomposition from the number of physical processors immediately frees one from artificial restrictions like "this program requires a cube of power of two processors". Only the logical entities need to have such a restriction, while the program continues to run on any arbitrary number of processors, with only a small loss of efficiency, if at all. That is, if the number of objects is not evenly divisible by the number of processors, and all the objects have an equal grain size.

Software engineering principles of cohesion and coupling are violated in MPI programs, viewed from a certain perspective: this comes about because of the processor-centric organization of an MPI program. Consider a program involving two parallel modules, say a "solids" and a "fluids" module. Typically, in an MPI program, the 17th partition (for example) of the *solids* domain and the 17th partition of the *fluids* domain are glued together on the 17th processor, simply because their respective partitioners called them "17th". Geometrically, they may not be connected and the corresponding computations may not exchange data with each other. In contrast, with object-based decomposition, one can partition the two domains independently into M and N pieces respectively, express the computation in terms of the $M + N$ objects, and allow the RTS to place the objects that communicate with each other together to the extent possible.

Concurrent composition of parallel modules is well supported by this model: since there are multiple objects on a processor, this model requires a message-driven scheduler on each processor. As a result, if two modules are invoked "simultaneously", idle time in one module can be automatically overlapped with useful computation from the other module.

The second category of benefits arises from the flexibility afforded by migratable objects, which can be exploited by an intelligent runtime system. Probably the most obvious of these is dynamic load balancing and is discussed in the next section. In addition, the RTS can shrink and expand the sets of processors allocated to a job by migrating objects accordingly. This can enable more powerful job scheduling where jobs don't need to languish in the queue waiting for the right number of processors to become vacant. An introspective RTS can also observe persistent communication patterns among objects and optimize communication by using appropriate algorithms, say for the collective operations such as all-to-all.

Automated checkpointing can be supported now with little extra effort by the programmer, since checkpointing is essentially "migrating" all the objects to the disk. The Charm++ and AMPI systems provide such automated checkpointing [31] in two separate modes: a traditional file-system based checkpointing, and a memory or local-disk based checkpointing. Migratable objects also enable an innovative fault tolerance scheme which (a) sends only the failed processors back to their checkpoints and (b) parallelizes the recovery by migrating the recovering objects from the failed processor to multiple other processors [4]. As a result, a job can continue to make progress even if the mean time between failures is less than the checkpointing period!

Message-driven scheduling leads to automatic adaptive overlap of communication and computation, as mentioned above. This also makes the programs more tolerant of latencies. This has been demonstrated somewhat dramatically in our multi-cluster results [20], where relatively tightly coupled benchmarks with time steps of a few milliseconds were able to tolerate multiple millisecond latencies between clusters located over a hundred miles apart. In addition, the fact that the scheduler can peek ahead at the queue can be exploited to prefetch the objects (and in some extreme cases, such as on IBM cell processors, their code) closer to the processor in

the memory hierarchy, which provides a significant performance boost in today's architectures.

In spite of this long list of successes, we believe that only the "low hanging fruit" has been exploited so far from this idea. Much more sophisticated runtime optimization strategies are yet to be developed. Scheduling challenges, such as those identified in [12] which require making complex trade-offs between parallelism and available memory along with load balancing, are examples of problems that will require further research.

3.2. Automate Load Balancing

Dictum 2: *Leverage the* principle of persistence *via measurement-based automatic load balancing.*

For most CSE applications, once a computation is expressed in terms of its "natural" objects, the computational loads and communication patterns *tend to* persist over time. We call this *the principle of persistence*. Note that this is simply an empirical observation, similar to the principle of locality. This principle often holds even when the application behavior is dynamic. Often, either the loads and patterns change slowly over time, or they change abruptly but infrequently. In either case, the recent past becomes a good predictor of near future, and this can be exploited by runtime load-balancing strategies that are enabled by migratable objects.

As an aside, it is worth pointing out that this methodology leads to a two-level load-balancing: application entities (data and computation) are already partitioned into chunks, and the load balancer only moves the chunks around the processors. As an example, an unstructured-mesh computation is traditionally re-balanced by calling a graph partitioner (such as parMETIS). This is typically a time consuming process. In contrast, the migratable objects method retains the original over-partitioning and only decides to migrate chunks, say because some chunks have gotten heavier. Since there are thousands of elements in each chunk, this is a much faster process. Moreover, the runtime strategies become domain independent: they apply to unstructured-mesh computations or gravity calculations based on the Barnes-Hut algorithm.

Since the runtime system is in charge of mediating communication and scheduling objects, it can automatically instrument and collect data on how long an object executes and how many bytes and messages are communicated between each communicating pair of objects. A load-balancing strategy, selected from a suite of strategies, then uses this database to create a new mapping of objects to processors, either from scratch or by refining the existing mapping.

Centralized load balancers collect the entire database on one processor and make decisions using a sequential algorithm. Since there are typically only tens (or hundreds) of objects per processor, the sequential bottleneck is not as strong a hurdle as one might expect. Indeed, we have shown that such balancers can be used to several thousand processors effectively. However, for much larger machines, this bottleneck must be eliminated. Fully distributed strategies [15, 30, 1], which we experimented with in early days, do not lead to a good quantity load balance. Instead, hierarchical and hybrid load balancers can combine the benefits of both kinds of strategies.

In our experience (at least currently), a suite of strategies, rather than a single super-strategy, is necessary, catering to differing needs of applications and parallel machines. Some strategies may trade-off communication costs for better load balance, or vice versa; others may minimize load-balancing time at the cost of quality, which trade-off is desirable for computations where the load changes rapidly.

Recent supercomputers, including BG/L, BG/P and Cray XT series, employ a three dimensional torus or grid interconnect, which necessitates another wrinkle for load-balancing strategies. Overall, such interconnects are beneficial because they can maximize link bandwidths due to shorter wires. Due to wormhole routing and its variants, the communication latency does not increase significantly with the number of hops; however the fraction of available bandwidth consumed by a message is proportional to the number of hops traveled. Thus, for communication

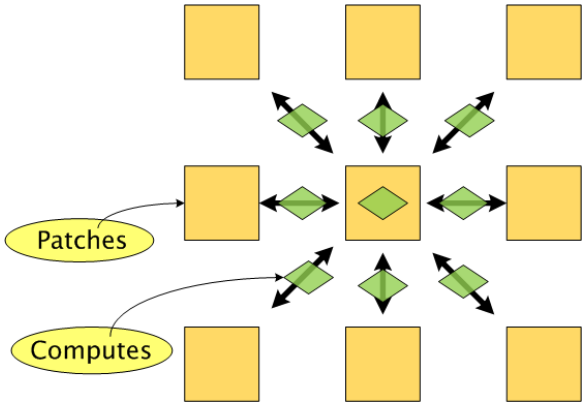


Figure 5. Object-based Parallelization for Molecular Dynamics

intensive applications, a load balancer must keep communicating objects on nearby processors to the extent possible.

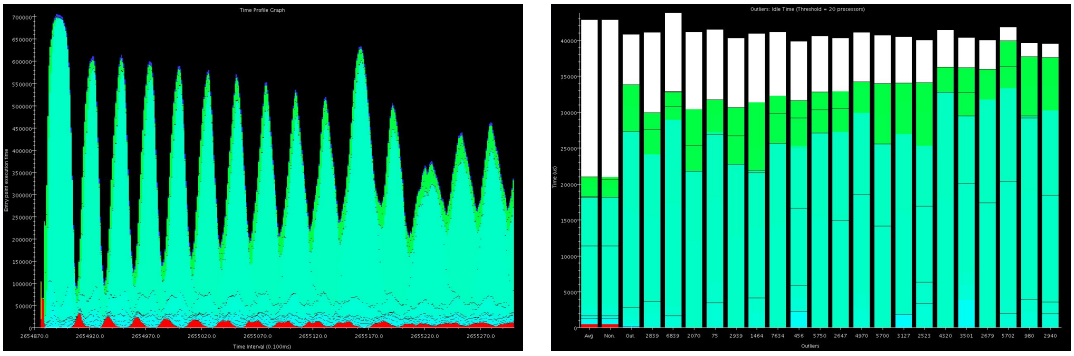
Automatic load-balancing faces many new challenges with the new generation of machines and applications. In addition to the load, memory footprint variation may have to be considered; heterogeneous processors including accelerators complicate the scenario further. However, it seems clear that for such large machines, an automated system (in the RTS) can do a better job than most humans can productively do.

3.3. Analyze Scalability of the Algorithm

Dictum 3: *Use asymptotic analysis and particularly isoefficiency analysis, to choose parallel algorithms that will continue to scale to newer generations of machines.*

When one uses a larger machine, one often wants to solve a larger problem. This early insight led to the notion of scaled speedups [9]. Isoefficiency analysis gives a precise definition of when an algorithm can be said to be scalable, and provides a quantification of scalability [8]. Consider an algorithm that is running on P processors with parallel efficiency η. If we increase the number of processors, can we also increase the problem size so that we can regain the same efficiency η? If so, the algorithm is said to be scalable. The rate at which the problem size (as measured by the total amount of computation, for example) must increase to maintain the same efficiency, as a function of the number of processors, quantifies the scalability.

Our early experience in designing the program NAMD illustrates this point. In the mid-1990s, when NAMD was designed, many other biomolecular simulation programs used either static atom decomposition or force decomposition methods. The former was a natural consequence of parallelizing an existing sequential code. Our isoefficiency analysis [18] of both methods showed that the communication to computation ratio will keep on increasing with increasing number of processors, independent of the number of atoms being simulated, thus rendering them non-scalable. This gave a solid basis for our decision to use spatial decomposition (which was also used by a few others, such as [25]). To generate additional parallelism in a flexible manner we hybridized this with force decomposition: a separate migratable object was created for calculating forces between atoms in each neighboring pair of cubes, and it could be placed by the load balancer on a third processor where neither of the interacting cubes are resident (see Figure 5). For the machines and molecular system sizes being considered at that time, the non-isoefficient methods were adequate; yet our design based in part on the isoefficiency analysis, has survived the test of time for over a dozen years.



(a) Time Profile View (b) Outliers View

Figure 6. Scalable Performance Visualization Tools

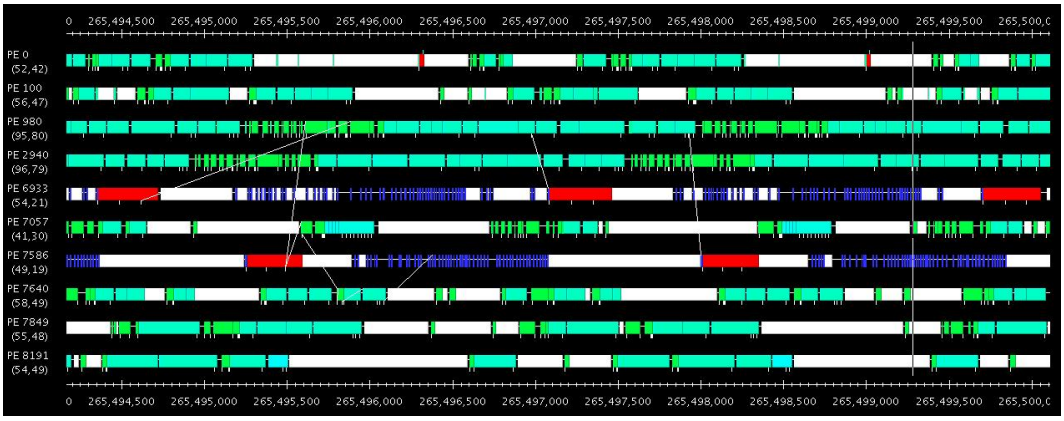


Figure 7. Timeline view of a NAMD run illustrating performance issues

3.4. Analyze Performance with Sophisticated Tools

Dictum 4: *Use scalable performance analysis tools.*

Performance of a parallel application on large modern supercomputers is a function of a large number of interacting factors. With sophisticated multi-module applications, deep communication stacks with adaptive and variable behaviors, and other complexities of a parallel architecture, it is not surprising that even the most well-designed applications require performance debugging (a.k.a. performance tuning). When one is dealing with, say, hundred thousand processors, running a fine-grained application that synchronizes every few milliseconds, even low-probability imbalances or bottlenecks will occur on some processor. Even on a thousand processors it is possible to view and analyze full traces, albeit barely so. But on larger machines, it becomes well-nigh impossible. A more automated solution is necessary.

There are two separate limits being breached. One is the perceptual limit of how much visualized data can a human brain comprehend; the second is the limit on the volume of trace data that needs to be stored on the file system, and the time needed to store and analyze it. For the former, a well engineered system that can process, sort and summarize the relevant performance data in perceptually comprehensible chunks is necessary. For the latter, an automated analysis, carried out in parallel, possibly at the end of the program run while the trace data is still in individual processor memories, will be necessary.

A relatively simple example illustrates this point. While debugging the performance of NAMD on 8,192 processors, which was below expectations, we examined the time-profile view

(Figure 6(a)), which shows the average processor utilization for a sequence of time intervals. The waxing and waning of utilization with each time step seemed like the signature of load imbalance; yet the dynamic load balancer was reporting that it had managed to bring the load imbalance within a few percent. An outlier analysis was performed to show the 20 most different processors (Figure 6(b)). This showed that the overloaded processors had a particular activity on them. This particular activity, an intermediate node of the tree-based multicast over a small set of processors, was assumed to be cheap. A more detailed time-line view of the most overloaded processors, and their communicating processors, showed the problem clearly (Figure 7). Not only were the intermediate modes of the spanning tree taking over 100 μs, which was much longer than expected, but multiple such intermediate nodes were being assigned to the overloaded processors. This went unnoticed by the load balancer because the spanning trees are formed, out of necessity, after the load-balancing phase, with respect to the new placement of objects. How this chicken-and-egg problem was solved is not pertinent here. The scalable outlier analysis presented a manageable picture to the programmer, which was instrumental in identifying this problem. Our current work [22], as well as that of a few other groups [23, 6, 26], is aimed at carrying out such analysis automatically at runtime, and storing only relevant summaries and a few representative detailed logs.

3.5. Model and Predict Performance Via Simulation

Dictum 5: *use an emulation-based program development and performance modeling system to prepare your application for a future parallel machine.*

The NSF funded Blue Waters system will be available for general use in Illinois in the middle of 2011. Although many details are still not public, it will have at least 200,000 processor cores, and will have multiple petaFLOPS peak performance. An application developer who starts tuning (or worse, developing) an application for this machine at that time will have a difficult time getting effective production runs during the first year. Simply doing paper and pencil modeling, or scaling it to the other parallel machines available to the NSF community may not be adequate. A similar challenge arises any time a new machine with substantially larger size, or substantially different architecture, than the prevailing machines is to be deployed. Further, even when a large machine is deployed, time available on the full machine for performance tuning may be limited and intermittent.

An approach towards a solution is provided by the *BigSim* project at Illinois. The idea leverages processor virtualization based on migratable objects mentioned earlier. One first runs the full-fledged application on a (relatively) smaller parallel machine in *emulation mode*. For example, a target application with a million objects (or AMPI threads) meant for a target machine of 100,000 processors can be run on a 10,000 processor machine, with the runtime system providing data structures that make it pretend to be the full size machine. During the emulation mode run, trace data is collected that records information about sequential execution blocks (SEBs), messages, and dependencies between them. This trace data is then fed to a parallel discrete event simulator along with architectural parameters of the target machine. The simulator run produces performance traces as if the program has been run on the target machine, which can then be analyzed using scalable performance analysis tools.

This basic outline shows the promise of the approach, but hides many technical challenges. Somewhat surprisingly, there exists a significant class of applications where such a 10-to-1 multiplexing of target processors to emulating processors does not cause a memory footprint problem. For other applications, an out-of-core execution scheme that leverages the peek-ahead capability of message-driven execution is necessary. Predicting execution time for SEBs on the target processors is another challenge. One can use a multi-resolution approach here, but at the highest resolution one may need to plug in (and/or interpolate) numbers from cycle-accurate simulators. Similarly, a multiple-resolution approach can be used for modeling communication

performance. Significant research is underway towards these objectives and an initial system is in place for application developers.

The technical details of solutions to these challenges are out of scope of this paper [32, 29]. However, it is important to understand what the utility of BigSim for petascale application development is: (1) the emulation is useful to test the basic scalability authentication data structures. (2) Even when the performance prediction is approximate, the performance traces obtained by the simulation are useful for identifying potential performance bottlenecks. (3) Application sensitivity to machine parameters (which may not be completely known yet) can be studied by running the simulation with varying machine parameters. For example, one can study how latency tolerant the application is. (4) runtime strategies such as load balancers can be tuned in an off-line mode by processing the trace data via the simulator. (5) off-line performance tuning using smaller version of the target machine becomes feasible.

4. Challenge of the Multicore Chip

Multicore chips are now the norm, and the number of cores per chip is expected to grow exponentially for next several years. Multiple such chips are present in individual nodes. Power considerations will keep clock frequencies low, possibly below 4 GHz. How should a petascale application developer adapt to these circumstances?

SMP nodes have been used for many years by the HPC community. However they are commonly used by setting up a separate MPI process for each processor. Hybrid models, typically using OpenMP within a node and MPI across, have been sporadically used but have not caught on. Experience suggests that a straightforward coding using OpenMP or pthreads within a node leads to worse performance than simply using MPI on each processor. False sharing, cost of locks, sloppy use of common locks for multiple unrelated critical sections, are some of the reasons for this. With significant effort, they can be overcome but typically only to equal the MPI performance.

Yet, there are benefits of using shared memory within a node that cannot be ignored, at least for some applications. In gravity calculations, a processor may request a set of particles from a remote processor. Since it may encounter a need for the same particles again during its tree walk, a processor level software cache is maintained. Using shared memory, one can maintain a *node-level* cache and reduce network traffic considerably, by avoiding duplicate requests.

We believe that a programming model that does not have the performance pitfalls of OpenMP or *Pthreads* is needed for exploiting shared memory within a node. Charm++, and MSA built on top of Charm++, have some of the features needed in such a model, but clearly further research is needed. For example, Charm++ objects can only access their own memory, in addition to node-level read-only data. With this pattern, and a careful memory allocation strategy, false sharing can be eliminated completely in user programs, nor are any locks needed in the user program. Of course, the RTS itself must reorganize its own data structures to limit or eliminate false sharing and reduce the penalty of locks and fences. However, once the RTS has been tuned, all the applications benefit uniformly. Similar benefits can accrue to users of frameworks such as Trilinos [10], which can also insulate the users from dealing with SMP performance issues.

Charm++ has included an SMP version for many years. However, it was typically not used by application users because of its performance. When we started examining the issue, we found that for messages up to several kilobytes in length, even messaging within the node was slower than using the network to send the message (see the top two curves in Figure 8)! The benchmark used to generate the data in Figure 8 has each processor core sending and receiving 12 messages in each iteration. With a series of optimizations in the RTS, which are instructive in themselves and will be described elsewhere, the performance was improved dramatically, even for short messages (the bottom curve in Figure 8 shows the performance after recent optimizations). For

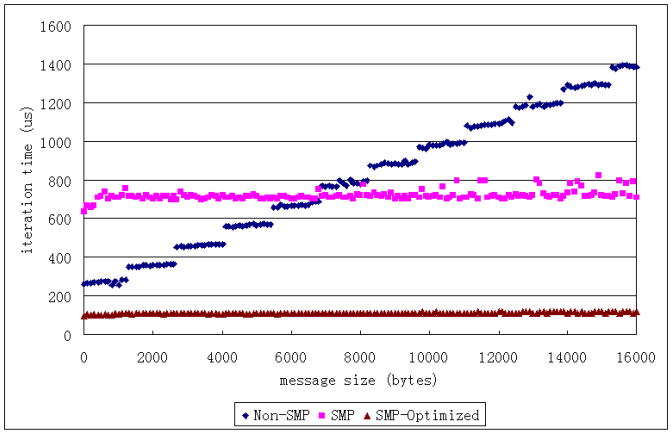


Figure 8. Performance of k-neighbor(multiple pingpong) iteration time

large messages, of course, the SMP mode performs better due to its avoidance of an extra copy. Note that in CHARM++, the ownership of a message sent to a remote object is transferred to it and therefore no copy at the runtime level is needed.

In addition to this, the RTS needs to incorporate new versions of load-balancing strategies, communication optimization strategies, and libraries that are "SMP aware" to derive full advantage of SMP nodes. Communicating objects should be preferably placed on the same node if they cannot be placed on the same processor, spanning trees for multicast groups need to be optimized, load balancers need to take multicast targets into account (especially for multicast over a set of objects whose number is significantly smaller than the total number of processors is, which is common in applications such as molecular dynamics) so as to prefer placing them on the same node. Many such strategies are currently being developed in our research, and in frameworks developed by other researchers.

5. Further Improvements in Productivity: New paradigms and Interoperability

The task of programming petascale machines for complex multi-module, multi-physics applications that use sophisticated algorithms and dynamic adaptation remains challenging, and can benefit from new programming models. Although systems such as Charm++ automate some aspects (e.g. load balancing) they do not raise the level of abstraction, especially for applications that don't need these features. How should we go about exploring and utilizing new programming models?

Since this is an open area of research, there are many opinions about it. However two meta-points seem evident: interoperability and need for exploration. The latter point simply means that we should not standardize "too soon", allowing many different approaches to be explored. The need for interoperability arises in part from the same desire. If we facilitate interoperation of multiple modules written in different parallel programming paradigms, it becomes possible to integrate one module based on a radical paradigm into an application consisting of many traditional-paradigm modules. This allows new paradigms to establish a "beachhead" within the HPC landscape. But the long-term reasons for supporting interoperability are deeper: it seems conceivable, or possibly even inevitable, that different paradigms would be suitable for coding different types of algorithms. A system that supports interoperability will therefore enhance productivity by allowing a suitable paradigm for each module.

Interoperability doesn't just mean the ability to call one module from another module written in a different paradigm. It means the ability to concurrently compose multiple patterns and modules. As explained in Section 3.1, this may require an approach based on over-decomposition

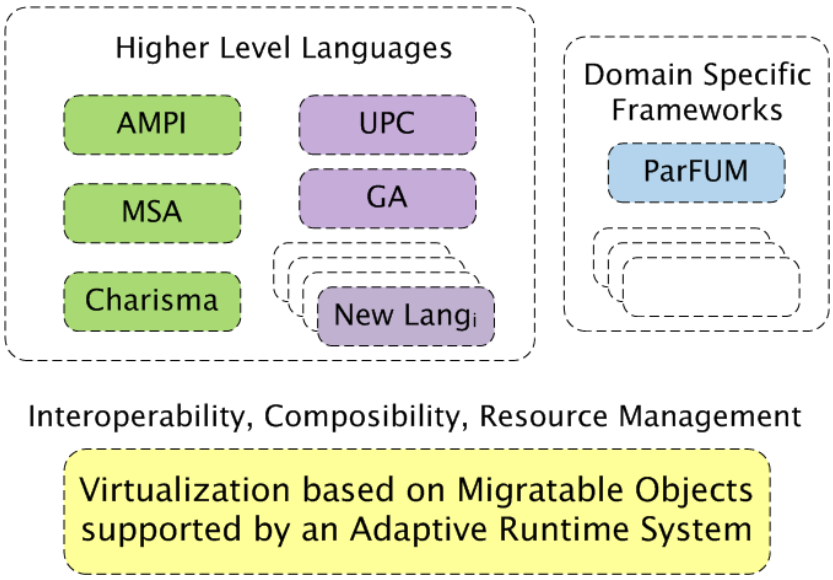


Figure 9. A complete programming solution with multiple interoperable paradigms.

and message driven execution.

A somewhat radical idea for raising the level of abstraction is to simplify parallel programming by giving up completeness! One can design a language/paradigm that is simple to use and debug applications with, yet cannot be used for expressing all parallel algorithms. As long as it is useful for expressing a significant subset of parallel programming patterns, it can be used to improve programmer productivity when combined with interoperability. Multiple such paradigms, interoperating with lower-level but complete paradigms such as MPI or Charm++, form a complete parallel programming solution (see Figure 9).

In our own work, we have developed two such paradigms. Both are deterministic in the sense that they do not permit expression of programs with race conditions. Multiphase shared arrays (MSA) [5] is a paradigm in which migratable threads can access user-partitioned global data arrays; the restriction is that each array must be accessed only in one of a few access modes, including read-only, exclusive-write, and accumulate. The access modes of an array may change across phases indicated by array-specific synchronization points. Charisma [11] is a language that supports static data flow communication only. The program consists of migratable objects (as in Charm++) that communicate with each other in predetermined patterns. These patterns are expressed in a script like notation, while the sequential code is expressed separately.

It seems clear that support for a global address space is a popular abstraction, though it need not be as general as cache-coherent shared memory. Co-array Fortran (CAF), UPC, and GA (Global Arrays) are examples of such paradigms. GA, for example, has been used in several widely used applications, such as NWChem [19]. It is necessary to find a method for supporting these (and related) abstractions with true (concurrently composible) interoperability, migratability and adaptive runtime systems, without losing the ability to leverage hardware support for one-sided (get/put) operations.

An answer along other dimension to the question of raising the level of abstraction is provided by domain-specific frameworks. An unstructured mesh framework, such as Sierra [28] or ParFUM [21], encapsulates commonly needed functionality within that domain into a reusable library, and provides an abstraction that simplifies expression of program modules within such domains. Since many CSE applications are covered by a relatively small number of data structures — unstructured meshes, structured meshes, adaptively refined structured meshes,

particles partitioned into uniform bricks/cubes or into hierarchical structures such as oct-trees — frameworks aimed at each of these can enhance productivity. Again, since the entire application may consist of multiple modules only some of which can use such a framework, it is necessary that such frameworks play well in an interoperable framework.

6. Conclusion

I have presented a viewpoint about what techniques are needed for programming upcoming parallel machines with performance of multiple PetaFLOPS. Admittedly, it is somewhat subjective and maybe even a parochial viewpoint, in that it is based on my research group's experience with parallel applications and presents a research agenda that overlaps significantly with my own.

Five techniques were identified and presented in the form of dictums: migratable-objects-based virtualization, load balancing based on principle of persistence, iso-efficiency analysis for scalability, scalable performance visualization/analysis, and emulation-based early application development. The paper then discussed challenges presented by multicore chips, and some approaches for overcoming them. Finally, it discussed methods for raising the level of abstraction in petascale programming, via multi-paradigm programming supported by an interoperable RTS that supports concurrent composition, via a collection of specialized, incomplete yet simple parallel programming paradigms, and via domain specific frameworks.

We are poised at a cusp in the history of computational science and engineering. The raw power of upcoming supercomputers is huge; the parallel programming challenge of exploiting this power is enormous; yet, if this challenge can be overcome, the societal benefits arising from science and engineering breakthroughs are even larger. A concerted effort by the HPC community in the next few years will make a difference.

References

[1] Abhinav Bhatele. Application specific topology aware mapping and load balancing for three dimensional torus topologies. Master's thesis, Dept. of Computer Science, University of Illinois, 2007. http://charm.cs.uiuc.edu/papers/BhateleMSThesis07.shtml.

[2] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[3] Eric Bohm, Glenn J. Martyna, Abhinav Bhatele, Sameer Kumar, Laxmikant V. Kale, John A. Gunnels, and Mark E. Tuckerman. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[4] Sayantan Chakravorty and L. V. Kale. A fault tolerance protocol with fast fault recovery. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

[5] Jayant DeSouza and Laxmikant V. Kalé. MSA: Multiphase specifically shared arrays. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[6] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, 24(1):163–173, January 1989.

[7] G. C. Fox et al. *Solving Problems on Concurrent Processors*. Prentice-Hall, 1988.

[8] A. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel & Distributed Technology*, 1(3), August 1993.

[9] J. L. Gustafson. Fixed Time, Tiered Memory, and Superlinear Speedup. In *Proceedings of the Fifth Distributed Memory Computing Conference*, volume 2, pages 1255 – 1260, 1990.

[10] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[11] Chao Huang and Laxmikant V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[12] Parry Husbands and Katherine Yelick. Multi-threading and one-sided communication in parallel lu factorization. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[13] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[14] L. V. Kale. Application oriented and computer science centered HPCC research. pages 98–105, 1994.

[15] L. V. Kalé, Milind Bhandarkar, and Robert Brunner. Load balancing in parallel molecular dynamics. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1457 of *Lecture Notes in Computer Science*, pages 251–261, 1998.

[16] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *Parallel Programming Laboratory Technical Report #95-02*, 1994.

[17] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *Parallel Programming Laboratory Technical Report #95-03*, 1994.

[18] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 1998.

[19] R.A. Kendall, E. Apra, D.E. Bernholdt, E.J. Bylaska, M. Dupuis, G.I. Fann, R.J. Harrison, J. Ju, J.A. Nichols, J. Nieplocha, T.P. Straatsma, T.L. Windus, and A.T Wong. *A.T. Computer Phys. Comm*, chapter High Performance Computational Chemistry: an Overview of NWChem a Distributed Parallel Application, pages 260–283. 2000.

[20] Gregory A. Koenig and Laxmikant V. Kale. Optimizing distributed application performance using dynamic grid topology-aware load balancing. In *21st IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[21] Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235.

[22] Chee Wai Lee, Celso Mendes, and Laxmikant V. Kalé. Towards Scalable Performance Analysis and Visualization through Data Reduction. In *13th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Miami, Florida, USA, April 2008.

[23] Allen Malony and Daniel Reed. Performance analysis, visualization with hyperview. *IEEE Software*, 5(3):26, May 1990. [Abstract].

[24] J. E. Moreira and V.K.Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41(3):303, 1997.

[25] S. J. Plimpton and B. A. Hendrickson. A new parallel method for molecular-dynamics simulation of macromolecular systems. *J Comp Chem*, 17:326–337, 1996.

[26] S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–331, Summer 2006.

[27] Seema Singh. Tata Hopes Its Supercomputer Is A Money Machine. page 18, June 2008.

[28] L. M. Taylor. Sierra - a software framework for developing massively parallel, adaptive, multi-physics, finite element codes. In *Presentation at the International conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, Nevada, USA, June 1999.

[29] Terry L. Wilmarth, Gengbin Zheng, Eric J. Bohm, Yogesh Mehta, Nilesh Choudhury, Praveen Jagadishprasad, and Laxmikant V. Kale. Performance prediction using simulation of large-scale interconnection networks in pose. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*, pages 109–118, 2005.

[30] Gengbin Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[31] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.

[32] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 78, Santa Fe, New Mexico, April 2004.