The Virtualization Approach to Parallel Programming: Runtime Optimizations and the State of the Art

L. V. Kalé
Parallel Programming Laboratory
University of Illinois at Urbana-Champaign
kale@cs.uiuc.edu

November 11, 2005

Abstract

For the past decade, we have been developing a parallel programming model based on virtualization. The basic idea is simple: let the programmer divide the work into a large number of chunks, mostly independent of the number of processors, and let the system map these entities to processors. This simple idea leads to an effective separation of concerns between the programmer and the runtime system (RTS), and empowers the RTS to carry out several tasks automatically that would normally require complex parallel programming skills. We describe the methodology, explain its advantages, and the success it has led to, including two Gordon Bell award nominations for "difficult to parallelize" applications.

1 Introduction

Parallel computing technology is poised to make a significant impact on society. The hardware technology itself is quite impressive: machines with tens of teraflops of peak performance already exist, at least one with over a hundred teraflops peak performance is on the anvil (Blue Gene/L), and one can realistically discuss building PetaFLOPS class machines. Scientific and engineering applications that will run on such machines tend to be dynamic, complex, and multidisciplinary. Programming such applications correctly, and with high execution efficiency on these machines is a challenging hurdle.

Specifying a parallel program can be seen as a multistage process. First, it involves deciding which actions to do in parallel, then deciding which processor will execute each action (and what data will be housed on each processor), followed by deciding the sequence in which each processor will execute the work assigned to it, and finally, expressing these decisions using primitives provided by the particular parallel machine on which the program will run.

Parallelizing compilers aim at automating the entire process. Research in this area has been intellectually stimulating, and has achieved considerable success. However, for large machines and complex applications it has proved to be inadequate. MPI automates the last step, thus providing a generic (and popular) machine independent parallel programming interface. However, in complex dynamic applications, MPI programming demands a significant effort from the programmer.

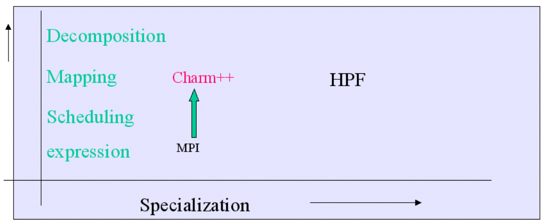

Figure 1: Technical Approach

The approach that we have been exploring is one level below the parallelizing compiler (see figure 1) Here, the programmer specifies only the decomposition into parallel parts, while the system maps the parallel parts to processors automatically. We think that this approach leads to an optimum division of labor between the programmer and the system.

In this approach, the number of parts a computation is broken into is typically not dependent on the number of processors (P), and is often much larger than P. One can think of these parts as virtual processors. Hence this approach can be loosely called virtualization (In the past, we have called it concurrent-object-based approach, message driven programming, or multi–partition decomposition).

Virtualization itself is not a new concept. Geoffrey Fox's 1986 textbook on parallel programming describes virtualization, for example (it was used to load balance the sharks-and-fishes application by dividing the domain into a large number of blocks, and sprinkling them across the processors randomly). The DRMS system [1] is another example of an approach based on virtualization. However, as will become clear in the paper, our approach (embodied in programming systems such as Charm++ and AMPI) can be thought of as virtualization++ : we support virtualization at the language and run-time level, and exploit it to the hilt to optimize application performance.

This paper summarizes the state of this approach which we have explored and applied for the past decade. One of the reasons the approach has been so successful is that it has always been developed in the context of multiple real applications [2]. One of the early applications of our methodology has been in molecular dynamics simulations of biological systems. This application, which has traditionally been difficult to parallelize, has been scaled to several thousand processors, and led to a Gordon Bell finalist position at SC2000 (At SC2002, newer results have led to another finalist selection for the award, and the award itself will be decided at SC2002). Several other applications, including simulation of atomic structures using QM/MM techniques, properties of materials, behavior of solid rocket boosters, and applications in computational cosmology are being developed using this approach.

In this paper, we will illustrate this approach, its embodiment in programming systems, and elaborate on the benefit it confers on parallel programming. We will illustrate these with examples from the application domains, and include performance data from a few applications.

2 Virtualization

The basic idea in virtualization is to let the programmer divide the program into a large number of parts, independent of the number of processors. The parts may be objects, for exam-

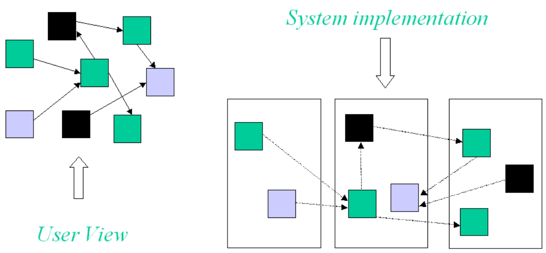

Figure 2: Object-based Parallelization

ple. The programmer does not think of processors explicitly (nor refer to processors in their programs). Instead they program only in terms of the interaction between these virtual entities. This is illustrated in Figure 2. Under the hood, the RTS is aware of processors and maps these virtual processors (VPs) to real processors, as it pleases. In particular, it can also change the mapping at runtime, without the user program having to specify it.

2.1 The Degree of Virtualization

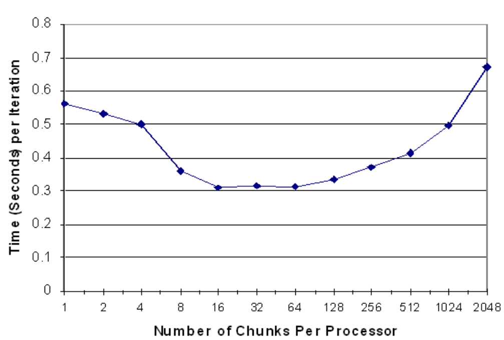

Figure 3: "Overhead" of Multipartitioning in an FEM application

How can one select the degree of virtualization independent of the number of processors (at least for most applications)? Essentially, the answer lies in the overhead associated with each virtual processor. Scheduling each execution (of a message or method invocation, for example) requires less than a microsecond on today's (2002) processors. Across-processor messages may have an overhead of 10 microseconds per message and a couple of nanoseconds per byte. The amount of computation done with each message must be "substantially larger" than this overhead to justify a lower granularity. Note that this definition does not depend on the number of processors, since the communication *overhead* (as opposed to latency) does not depend on the number of processors. Other factors influencing the decision are cache effects. With beneficial result of smaller objects on cache performance, one actually gets better performance with a higher degree of virtualization, instead of being dominated by higher overheads, as shown in Figure 3.

Somewhat more negative are situations where large ghost regions are required around each virtual processor. Here, the degree of virtualization must be constrained by the memory overhead of the extra space also. This situation can be mitigated somewhat by using techniques for storing ghost regions transiently in dynamic storage, and using schemes to fuse the "touching" objects that happen to reside on the same processor during a load balancing era.

We next describe Charm++, a C++ based programming system that supports this programming model.

2.2 Charm++

The basic unit of parallelism in Charm++ is a C++ object containing methods which may be invoked asynchronously from other such objects (which may reside on other processors). Such objects are called chares in Charm++.

Although Charm++ supports important constructs such as specifically shared variables, prioritized execution, and object groups, from the point of view of this paper, the most important construct in Charm++ is an object array ([3, 4]).

A chare-array has a single global ID, and consists of a number of chares, each indexed by a unique index within the array. Charm++ supports 1D through 6-dimensional arrays (sparse as well as dense). In addition, users may define a variety of other index types on their own (for instance, an Oct-Tree might be represented by a chare array, where the index is a bit-vector representing the position of a node in the tree).

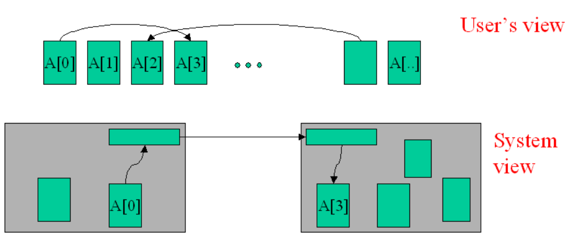

Figure 4: Object Arrays

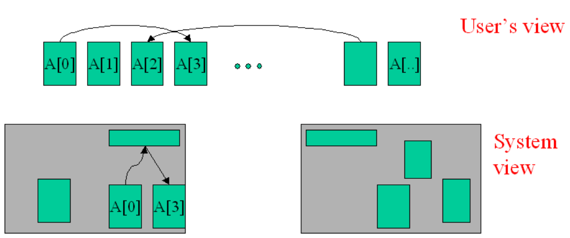

Figure 5: Object Arrays

As shown in Figure 4, the user thinks of the array as a set of objects, and the code in these objects may invoke methods of other array members asynchronously. However, the system mediates the communication between the objects. As a result, if the object moves from one processor to another, other objects don't need to know this. The system has automatic mechanisms that efficiently forward messages when needed, and cache location information so as to avoid forwarding costs in most situations. ([4]). In addition to method invocation, chare arrays support broadcasts ("invoke this method on all live array members") and reductions, which work correctly and efficiently in the presence of dynamic creation and deletions of array members as well as their dynamic migration between processors.

Chare Kernel, the C based version of Charm, was developed before 1989 ([5]), while the C++ based version (Charm++) was created in 1992. ([6]). The C based version (with its notion of object encoded as a global pointer, with a function to execute on the destination processor) has similarities to nexus ([7]), except that nexus entities were not migratable. Active messages and message driven execution in Charm++ are quite similar, althought the original notion of active messages was interrupt based (and akin to an implementation of inexpensive interrupts on CM-5). Also, message driven objects of Charm++ are similar to the Actors model ([8]). However, Charm++ arose from our work on parallel Prolog, and the intellectual progenitors for our work included the RediFlow project of Bob Keller ([9]), Our approach can also be considered a macro-data-flow approach. ([10]). Other research with overlapping approaches include work on Percolation and Earth multi-threading system [11], work on HTMT and Gilgamesh projects [12], and the work on Diva [13].

2.3 Adaptive MPI (AMPI)

In 1994, we conducted extensive studies ([14]), and demonstrated that Charm++ had superior performance and modulartity properties compared with MPI. However, it became clear that MPI is the prevalent and popular programming model. Although virtualization in Charm++ was useful, the split phase programming engendered by its asynchronous method invocation was difficult for many programmers, especially in Science and Engineering. MPI's model had certain anthropomorphic appeal. The processor sends a message, waits for a message, does some computation, and so on. The program specifies the sequence in which a processor does these things.

Given this fact, we decided to take the virtualization aspect of the Charm++ model, and apply it to MPI, so MPI programs can be easily "converted" to our framework. The resultant system is called AMPI. In AMPI, as in Charm, the user programs in terms of a large number of MPI "processes", independent of the number of processors. AMPI implements each "process" as a user-level light-weight and *migratable* thread as shown in Figure 6. This should not be confused with a PThreads style MPI implementation. We can have tens of thousands of such MPI threads on each processor, if needed, and the context switching time is of the order of 2-5 microseconds on today's machines. Migrating user-level threads, which contain their own stacks and heaps, and which may contain pointers to stack and heap data, is technically challenging, but has been efficiently implemented ([15]), based in part on the isomalloc technique developed.([16]).

Migrating existing codes to AMPI requires some simple, mechanical transformations to code (because global variables cannot be shared between two threads running on the same processors). To facilitate migration of application codes in Fortran, we developed a source-to-source translator called AMPIzer ([17]) based on the Polaris compiler framework ([18]). AMPI has been used to parallelize several applications, including codes at the DOE supported Rocket Simulation center at Illinois, and early versions of a Computational Cosmology code (which was converted to AMPI within one afternoon, sans the optimizations that followed).

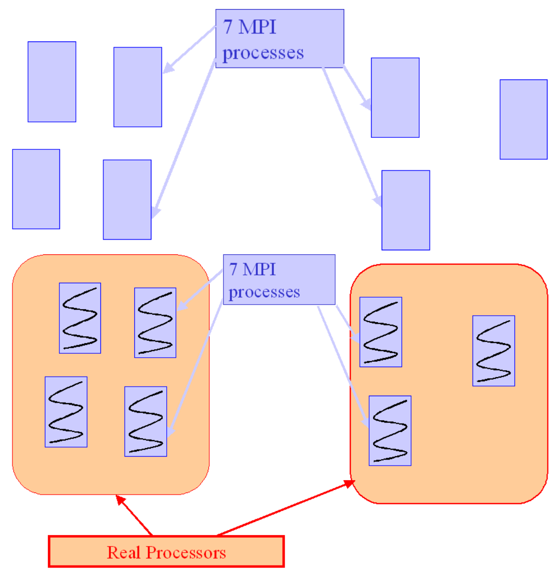

Figure 6: AMPI implements MPI processes as user-level migratable threads

3 Benefits of Virtualization

Charm++ and AMPI are only examples of the virtualization approach. Just as AMPI "virtualizes" the MPI model, one can imagine creating virtualized programming models from existing programming models (such as GPShmem, Global Arrays, and UPC, for example). In this section we outline a large number of benefits that acrue from virtualization. For ease of discussion, we have grouped them into benefits due to:

1. Better software engineering (3.1)
2. Message-driven execution (3.2)
3. Ability to dynamically map work to processors (3.3)
4. The principle of persistence (3.4)

3.1 Software Engineering Benefits

In software engineering, cohesion and coupling are important concepts. Good software engineering practice requires that program entities (codes/subroutines/data) should be coupled only when there is a logical connection between them. MPI's processor-centric model often leads programmers to violate these principles. The simplest examples are in specifying the number of processors to be used. A structured-grid application may require a uniform 3D decomposition. An MPI program then requires that it be run on a cubic number of processors (and often, a power-of-two cubic number for other reasons!). In contrast, in AMPI or Charm++, the program may divide its work into logical entities (a cubic number of virtual processors, or oct-tree nodes containing fewer than K particles, for example), and the physical number of processors is unimportant. Since there will typically be tens of VPs on a physical processor (and as described later, automatic load balancing will "pack" them uniformly onto processors), the performance will be fine, even when no dynamic behavior is expected.

As another example, consider an early version of the rocket simulation application: it consisted of 2 parallel modules: RocFlo (Fluid simulation of the burning gases in the Rocket interior) and RocSolid (Structural dynamics of the solid fuel). These were (derived from) independently developed codes. Since different mesh decomposition

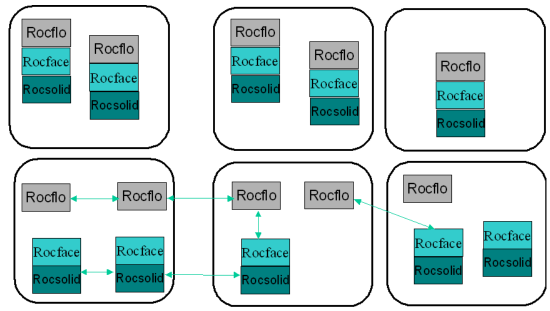

Figure 7: Rocket simulation via virtual processors

programs were used by each module, the portion of space simulated by Rocflo on processor i had no logical connection with that simulated by RocSolid on processor i. However, an MPI implementation required them to be fused together on each processor (Figure 7. An AMPI implementation, on the other hand, provided each module with its own set of virtual processors, and allowed for communication across them by supporting inter-communicators across multiple `MPI_COMM_WORLD`s. Among other benefits, this allows the number of pieces of RocFlo to be determined independently of that of RocSolid, and the RTS is able to bring together (on one processor) pieces of Rocflo and Rocsolid that directly interact (because they are physically abutting, for example).

3.2 Message Driven Execution

Since there are multiple virtual processors housed on each processor, it is necessary to have a dynamic scheduler on each processor, as shown in Figure 8. The scheduler works with a pool of "messages". It picks the next message from the pool (in accordance with priorities associated with messages, if any), identifies the charm++

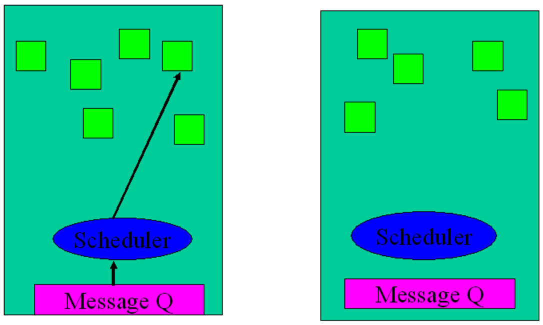

Figure 8: Message Driven Execution

object it is destined for, and invokes the designated method on it. The method runs to completion, producing other messages (method invocations) for objects on this or another processor. The scheduler then continues by selecting another message. If the message is for an AMPI thread, it is inserted in the queue of received messages, and the thread is awakened (i.e. marked as ready, and inserted in the scheduler's queue).

Thus, no object (or VP) can hold the processor idle while it is waiting for its message. Instead the object that has a message waiting for it is allowed to continue. This execution style is called message-driven execution (MDE), and leads to several key benefits.

3.2.1 Adaptive Overlap

One of the issues in MPI programs is that of overlapping communication with computation. When the program waits at a receive statement, we'd like the message it is waiting for to have already arrived. To achieve this, one tries to move `sends` ahead and `receives` down, so that between sends and receives one can get some computation done, giving time for communication to complete. When multiple data items are to be received, one has to make guesses about which will arrive sooner, or use wild-card receives, which often break the flow of the program ([14]).

With MDE, adaptive overlap between communication and computation is automatically achieved, without programmer intervention. The object or thread whose message has already arrived will be automatically allowed to continue, via the message-driven scheduler.

This advantage is all the more stronger when one considers multiple parallel modules (Figure 9). A, B and C are each parallel modules spread across all processors. A must call B and C, but there is no dependence between B and C. In traditional MPI style programming, one must choose one of the modules (say B) to call first, on all the processors. The module may contain sends, receives, and barriers. Only when B returns can A call C on each processor. Thus idle time (which arises for a variety of reasons including load imbalance, and critical paths) in each module cannot be overlapped with useful computation from the other, even though there is no dependendence between the 2 modules.

In contrast, with MDE, A invokes B on each (virtual) processor, which computes, sends initial messages, and returns to A. A then starts off module C in a similar manner. Now B and C interleave their execution based on availibility of data (messages) they are waiting for. This automatically overlaps idle time in one module with computation in the other, as shown in the Figure. One can attempt to achieve such overlap in MPI, but at the cost of breaking the modularity between A B and C. With MDE, the code in B doesn't have to know about that in A or C, and vice versa.

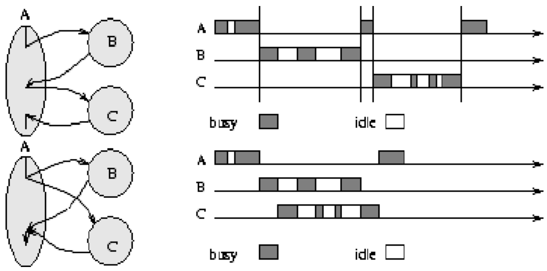

Figure 9: Modularity and Adaptive Overlapping

3.2.2 Predictability of Execution

The scheduler in Figure 8 knows which objects (and which methods) it will execute next. Looking at the next k items in the queue it can predict what the next k objects will be. This ability to predict next code and data to be accessed can be very beneficial for the RTS. For example, we can now automatically support out-of-core execution ([19]). The scheduler's queue is used to prefetch the next k objects from the disk into memory (replacing others as necessary).

In processors-in-memory (PIM) style PetaFLOPS class machines of the future, this predictability can be used to prefetch data (and even code) into a cache or RTS controlled SRAM.

3.3 Dynamic and Flexible Mapping to Processors

A virtualization based parallel programming system is supported by a run-time system that provides mechanisms and strategies for migrating the virtual processors among the real processors. (As discussed earlier, both Charm++ and AMPI share the same run-time system and provide migration mechanisms). This capability leads to several benefits including effective use of clusters that include desktop machines, changing the set of processors used by a job at run-time, and supporting automatic checkpointing. Automatic dynamic load balancing also follows from this capability, but is discussed in the next subsection since it benefits even more from another property, the principal of persistence, discussed in the next section.

3.3.1 Flexible Mapping on Clusters

Consider a parallel application running on a cluster that includes some desktop machines. The owner of a desktop might have allowed the application to be run on their machine only as long as they are not using it. When the owner starts using the workstation, a program based on virtual processors simply migrates them to the remaining processors, and continues execution uninterrupted. ([20]).

Even if the desktop owner does not insist on the parallel application vacating his workstation, the application may still suffer significant performance losses, when the owner starts running a computationally significant application of their own on that one processor. In a 16 processor cluster, the parallel application will lose half of one-sixteenth, or about three percent of the compute power. However, because of the dependencies between application components the entire application can slowdown to about 50 percent of its original speed. With migratable objects, runtime system can migrate about half of the objects (actually, number of objects equaling half of the computational load on that processor) to other processors, and restore performance to almost 97 percent of the previous level ([20])

Often, a cluster is built by incrementally adding new machines to it. Under these conditions, the machines tend to be of different speeds. The virtualization based run-time system sim-

Phase	16P3	16P2	8P3, 8P2 w/o LB	8P3, 8P2 w/ LB
Fluid update	75.24	97.50	96.73	86.89
Solid update	14.86	52.50	52.20	46.83
Pre-Cor Iter	117.16	150.08	149.01	133.76
Time Step	235.19	301.56	299.85	267.75

Table 1: Performance of Rocket Simulation on Turing Cluster

ply measures the speed of processors at the beginning (or reads it from a configuration file), and adjusts the allocation of objects to processors in accordance with their speeds. Table 1 shows this on an old cluster consisting of Pentium 2 and Pentium 3 processors, running the rocket simulation application. As can be seen, AMPI achieves the optimal performance using the mix of processors.

3.3.2 Changing the Set of Processors Assigned to a Job

Probably one of the most dramatic uses of virtualization is in changing the number of processors used by application at run-time. Consider a parallel machine with a thousand processors running a job on 800. If a new job that needs 400 processors arrives, it will have to wait for the first job to finish, while the system wastes 200 processors. With virtualization, our run-time system is capable of shrinking the first job to 600 processors, by migrating objects away from the 200 additional processors needed by the second job. Further, when the second job finishes, the system can expand the set of processors allocated to the first job to the original 800 again. We have already demonstrated a cluster scheduler ([21]) that makes use of this capability to optimize its utilization, and response time. This is especially useful for interactive parallel jobs, which are slowly becoming popular (for example, interactive molecular dynamics, or cosmology analysis and visualization).

This capability is also useful for jobs that themselves know when they need more or fewer processors, based on their application's dynamic structure, as well as jobs that are perpetually "hungry" for more processors, which are willing to run in the background at a lower priority.

3.3.3 Automatic Checkpointing

Checkpointing is simplified considerably for applications using the virtualized run-time system. After all, checkpointing is nothing but migrating the virtual processor to the disk, and we already have support for migration in the run-time system. What is more, a job checkpointing on 2048 processors can be restarted on 1900 processors, or 2500 processors if necessary. This is possible because restarting a job simply involves migrating the objects from disk to the available set of processors. Of course, several run-time structures must be properly restored, but this is done by the run-time system writer once. ([22]).

3.4 Principle of Persistence

The *Principle of Persistence* is a heuristic that can be thought of as the parallel analog of the principle of locality that we "discovered" while examining the success of our runtime system for dynamic load balancing. As you will see, this is an obvious principle.

Principle of persistence: Once an application is expressed in terms of its natural objects (as virtual processors) and their interactions, the object computation times and communication patterns (number and bytes of messages exchanged between each communicating pair of objects) *tend to* persist over time.

This heuristics holds for most parallel applications, including many with dynamic behavior. For example, some applications such as those using adaptive mesh refinement may involve abrupt but infrequent changes in these patterns. Other applications such as molecular dynamics may involve slow changes in these patterns. In either case, the correlation between recent past and near future, expressed in terms of interacting virtual processors, holds. In rare cases, applications may involve rapid and large changes, which can still be handled by our RTS as best as the application programmer will be able to handle it, by migrating work away from busy processors as they are detected. (The obviousness of the principle arises from the fact that most parallel applications are iterative in nature, and the physical system being simulated change gradually, due to numerical constraints).

The real benefit of this principle is that it allows for measurement-based load balancing and optimization algorithms (such as for communication libraries) that can adapt to application behavior.

3.4.1 Dynamic Load Balancing

Armed with the principle of persistence, the Charm RTS employs a measurement based load balancing scheme. It instruments the RTS (without user intervention) to collect statistics on each object's computational load and communication patterns, to build a load database. A suite of dynamic load balancing strategies (some centralized, some fully distributed, some incremental and others periodic and comprehensive) can tap into this database to make decisions about when and where to migrate the objects.

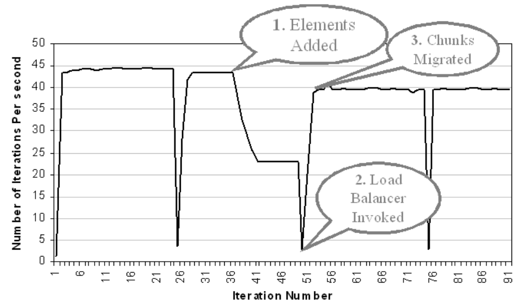

Figure 10: Automatic Load Balancing in Crack Propogation

An example of dynamic load balancing in action is shown in Figure 10. An FEM-based structural dynamics application is running on a parallel computer. The y-axis shows its throughput (i.e. the number of iterations completed per second), while the x-axis shows the iteration number. At timestep 37, the application experiences an adaptive event (some new elements are added as a result of a crack propagating into the domain). At the next load balancing event (the system is using a periodic load balancer), the performance is brought back to almost the original level, by migrating objects precisely from the overloaded processors to others, based on the measurement of load each object presents.

(This experiment used an artificial refinement event. Real application will take much longer to induce the refinement and insertion of new elements).

3.4.2 Communication Optimizations

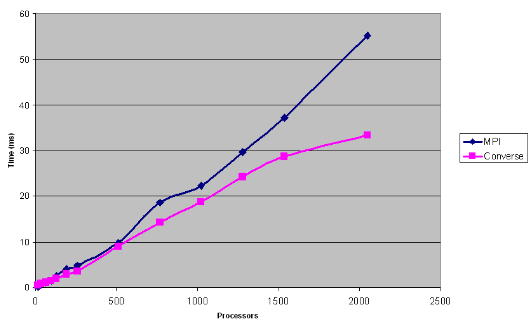

Figure 11: All to all on Lemieux

Object communication patterns provide another fertile ground for optimization by the RTS. For example, we have developed a suite of libraries that include several different algorithms for common communication alogirthms among objects (e.g. each-to-all-individualized messages). The library provides a common interface for these operations. However, at runtime, based on its observation of the number of bytes, number of objects, and number of processors involved (among other things), the library may switch from one algorithm to another, and tune an individual selected library's parameters to the prevailing communication conditions. It can also take into account concurrently executing communication operations (say from other modules) in making these decisions. We have explored this approach in ([23]).

A recent example of optimizing performance for an all-to-all operation is shown in Figure 11.

If the user provides control-points ("knobs") to the RTS to control the degree of virtualization, the system can even optimize the degree of pipelining. Manual optimization along these lines is illustrated in Figures 12 and 13, which arose in a recent collaborative work with QM/MM applications Scientists (Glenn Martyna and Mark Tuckerman, authors of the PinyMD ([24]). This code involved multiple concurrent 3D FFTs and the performance is clearly affected by the degree of pipelining (number of virtual objects per real processor) as seen in the snapshots from our performance visualization tool projections.

Figure 12: Pipelining

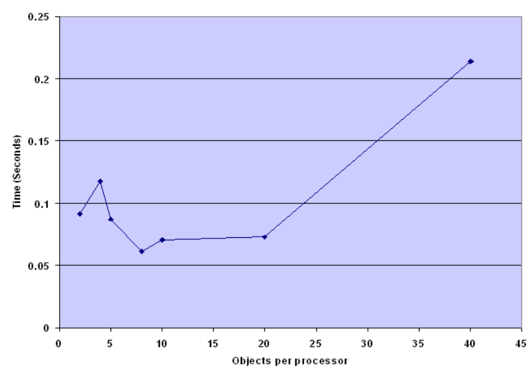

Figure 13: Effect of Pipelining

4 Case Studies

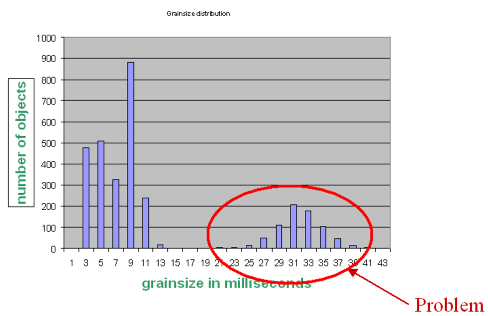

Figure 14: Grainsize Analysis

One of the early applications developed using Charm++ is NAMD ([25, 26]). It is a production quality molecular dynamics program for biomolecules such as proteins, DNA and cell membranes, as well as water molecules. Applications need to run for millions of timesteps, each simulating a femtosecond in the life of the molecular system, which typically consists of 10,000 to several hundred thousand atoms. This makes it a hard problem to scalably parallelize. In the Charm++ based implementation, NAMD creates many computational objects (e.g. 30,000 objects of about 1 msec work each, in a recent run). The load balancers maps these objects to processors periodically, taking several complexities of the communication patterns involved into account.

The details of the application parallelization ([25, 26]) are beyond the scope of this paper, but we note here that object granularity often becomes very important in such applications. NAMD completes each timestep in 10-20 msecs on the largest processor configurations we have used. So, no single object can be allowed to be large. Analysis tools in 2000 identified (See Fig. 14) some objects with "large" execution times, which were eliminated by increasing the degree of virtualization in a systematic way, to lead to a speedup of about 1250 on 2000 processors (See Fig. 15), with 40 GF performance, leading to a finalist position for the Gordon Bell (special category) award at SC2000. Current performance of the application is of the order of 0.65 TeraFLOPS (See Fig. 16), on about 2000 processors of the HP/Compaq LeMieux machine at Pittsburgh Supercomputing Center (which consists of about 3000 processors), even though the newer version includes Particle-Mesh Ewald, an

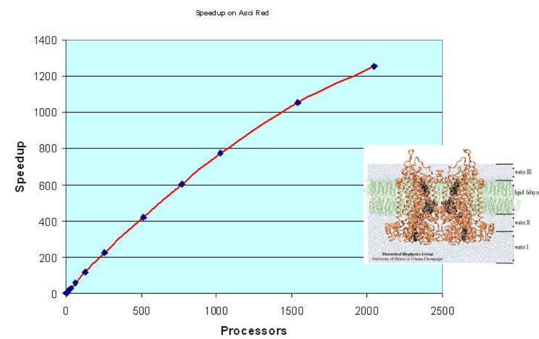

Figure 15: Improved Performance Data

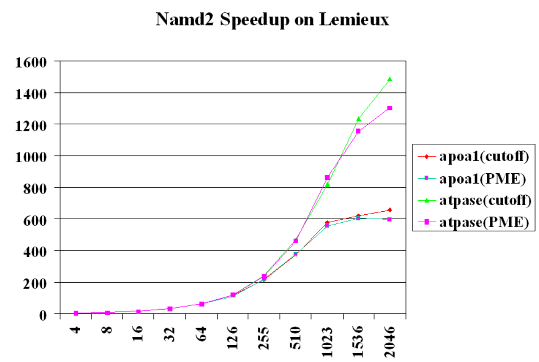

Figure 16: Namd2 Speedup on Lemieux

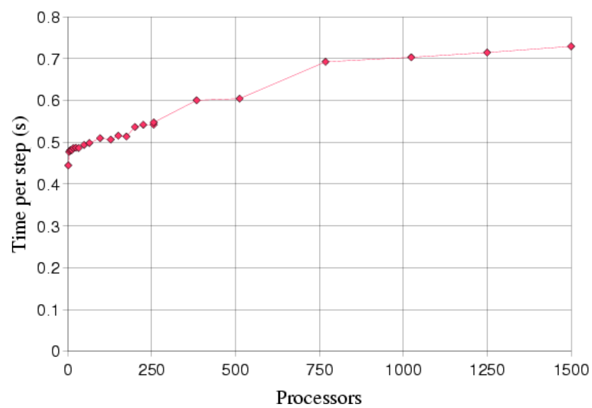

Figure 17: Time per step for a scaling parallel benchmark, with a fixed 65,536 polygons per processor.

alogrithm that requires 3-D FFT.

Another example of the utility of the virtualization is provided by Orion Lawlor's work on collision detection [27] (or "contact" detection in the structural dynamics parlance). We concentrated on the problem of contact detection, leaving aside the domain dependent problem of how to deal with the contact to the application program. Virtual processor based collision detection algorithms divide space into voxels, and sends each geometric surface object (say a triangle) to all the voxels it intersects with. Although many voxels that are not near the boundary of potentially colliding surfaces get no triangles at all, the algorithm is able to scalably detect collisions at the speed of a few microseconds per triangle involved. Performance data from a recent paper [27] is shown in Figure 17.

5 Summary and Future Issues

We described the virtualization model, its current state of art in terms of Charm++ and AMPI systems, and applications, and illustrated the benefits of this approach. Virtualization leads to better software engineering for parallel programs because it frees one from organizing applications in terms of the number of physical processors. Virtualization support leads to message driven execution, which promotes modularity, adaptively overlaps computation and communication and engenders execution predictability. Virtualization allows the RTS to assign and reassign virtual processors (objects or AMPI threads) to processors at runtime, leading to better performance of time-shared clusters, shrinking-and-expanding the sets of processors assigned to a job, and support for automatic checkpointing. *The Principle of Persistence* can be observed to hold for most parallel programs when expressed in terms of their natural pieces (as virtual processors). This leads to measurement based load balancing, and learning algorithms, including communication libraries that can tune their behavior to the application's dynamic and evolving characteristics. We alluded to a few applications being developed using this approach and the excellent performance results we have obtained so far.

The promise of the virtualization model is yet to be fully realized. More research is needed on strategies (including scalable load balancers, alternative communication libraries, compiler support for optimization and ease of use and more aggressive use of prefetching, for example).

Some directions of future research we are exploring include:

1. *Domain specific frameworks:* Referring back

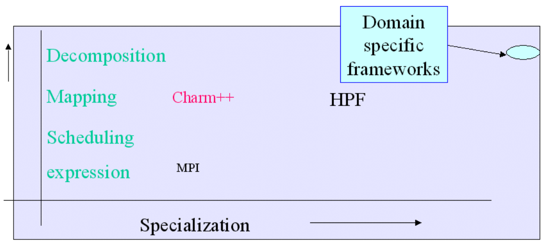

Figure 18: Component Frameworks

to the "parallel programming approaches" diagram we started with (See Fig 18), we believe that to go a higher level from Charm++/AMPI, the most fruitful way is to go in the horizontal direction, to more specialization (rather than vertically to automatic parallelization for general purpose sequential programs). To this end, we are building frameworks for structured and unstructured grid computations [28], AMR, and particle based computations, built on top of the virtualization model of Charm++.

2. *Components and Orchestration:* Ability to connect independently developed parallel components quickly and efficiently is crucial for modular development of complex applications. The CCA architecture ([29]) is a major effort in this direction. The virtualization model fits very well in this context, because of its flexibility in supporting "loose" communication among its components (Figure 5). We have started building a framework called Charisma ([30]) to this end. Further we are developing "orchestration languages" that express the control and data flow of a multi-component application, spanning over several sets of virtual processors across multiple independent modules. Further, cutting across all the set of applications we are collaborating on, we are building a "standard library" of virtualized parallel components.

3. *PetaFLOPS and PIM computers:* The virtualization model is especially suited for PIM based and other massively parallel machines being contemplated today. We are building a programming environment for the Blue Gene (BG/L) Machine, which is likely to have 64K dual-processor nodes, and for a conceptual design for Blue Gene/Cyclops, the million processor machine, with many simple mutlithreaded processors on a chip. We expect to collaborate with IBM and other researchers to develop and test such programming models and applications, using a emulation/simulation system we have built [31, 32].

4. *Virtualizing other models.:* To allow for modules written in different parallel programming languages to coexist, we built a framework called Converse [33]. We'd like to invite other researchers to use Charm++ and its underlying Converse framework to virtualize their languages, and experiement with multi-paradigm parallel programming, where each module is written in the paradigm best suited for its algorithms, and pre-written libraries can be used irrespective of the parallel language (or paradigm) they are written in. We are beginning a small effort of our own in this direction. (http://charm.cs.uiuc.edu/research/converse).

Virtualization based approaches are mature, and ready for the next generation application machines and applications. Further collaborations to explore this model jointly, as a community, will be very fruitful.

Acknowledgements

The research summarized here is funded in part by several grants from NSF, DOE and NIH over the past 10 years.

The system work in developing the Charm++ approach was done with several generations of graduate students. Specifically, for brevity, I will mention recent and current graduate students centrally involved with the system: Sanjeev Krishnan, Josh Yelon, Milind Bhandarkar, Gengbin Zheng and Orion Lawlor. Many other students at the Parallel Programming Laboratory at Illinois have contributed to the research and helped in writing this paper.

References

[1] 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.

[2] L. V. Kale. Application oriented and computer science centered HPCC research. In *Proceedings of Developing a CS Agenda for High-Performance Computing*, pages 98–105, March 1994. Position Paper.

[3] Sanjeev Krishnan and L. V. Kale. A parallel array abstraction for data-driven objects. In *Proceedings of Parallel Object-Oriented Methods and Applications Conference*, Santa Fe, NM, February 1996.

[4] O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, pages 21–29, Stanford, CA, Jun 2001.

[5] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, August 1990.

[6] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[7] Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, May 1994.

[8] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[9] R.M. Keller, F.C.H. Lin, and J. Tanaka. Rediflow Multiprocessing. *Digest of Papers COMPCON, Spring'84*, pages 410–417, February 1984.

[10] Arvind and S. Brobst. The Evolution of Dataflow Architectures: From Static Dataflow to P-RISC. *International Journal of High Speed Computing*, 5(2), 1993.

[11] H. Hum. A design study of the earth multiprocessor, 1995.

[12] G. Gao, K. Theobald, A. Marquez, and T. Sterling. The htmt program execution model, 1997.

[13] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping irregular applications to DIVA, A PIM-based data-intensive architecture. In *Proceedings of the High Performance Networking and Computing Conference (SC'99)*, 1999.

[14] A. Gursoy. *Simplified Expression of Message Driven Programs and Quantification of Their Impact on Performance.* PhD thesis, University of Illinois at Urbana-Champaign, June 1994. Also, Technical Report UIUCDCS-R-94-1852.

[15] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[16] R. Namyst and J.-F. Méhaut. PM^2: Parallel multithreaded machine. A computing environment for distributed architectures. In *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*, pages 279–285, Amsterdam, February 1996. Elsevier, North-Holland.

[17] Karthikeyan Mahesh. Ampizer: An mpi-ampi translator. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champiagn, 2001.

[18] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of 7th International Workshop on Languages and Compilers for Parallel Computing*, number 892 in Lecture Notes in Computer Science, pages 141–154, Ithaca, NY, USA, August 1994. Springer-Verlag.

[19] Neelam Saboo and L. V. Kalé. Improving paging performace with object prefetching. Technical Report 01-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, July 2001.

[20] Robert Brunner, Laxmikant Kalé, and James Phillips. Flexibility and interoperability in a parallel molecular dynamics code. In *Object Oriented Methods for Interoperable Scientific and Engineering Computing*, pages 80–89. SIAM, October 1998.

[21] Laxmikant V. Kalé, Sameer Kumar, and Jayant DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[22] Sameer Paranjpye. A checkpoint and restart mechanism for parallel programming systems. Master's thesis, University of Illinois at Urbana-Champaign, 2000.

[23] Sanjeev Krishnan. *Automating Runtime Optimizations For Parallel Object-Oriented*

Programming. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[24] Tuckerman ME, Yarne DA, Samuelson SO, and GJ Martyna. Exploiting multiple levels of parallelism in Molecular Dynamics based calculations via modern techniques and software paradigms on distributed memory computers. *Comput. Phys. Commun.*, 128:333, 2000.

[25] 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*, 151:283–312, 1999.

[26] R. Brunner, J. Phillips, and L.V.Kalé. Scalable molecular dynamics for large biomolecular systems. In *Proceedings of SuperComputing 2000*, 2000.

[27] Orion Sky Lawlor and L. V. Kalé. A voxel-based parallel collision detection algorithm. In *Proceedings of the International Conference in Supercomputing*, pages 285–293. ACM Press, June 2002.

[28] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[29] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *Processings of the 10th SIAM Converence on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 2001.

[30] Milind A. Bhandarkar. *Charisma: A Component Architecture for Parallel Programming*. PhD thesis, Dept. of Computer Science, University of Illinois, 2002.

[31] Neelam Saboo, Arun Kumar Singla, Joshua Mostkoff Unger, and L. V. Kalé. Emulating petaflops machines and blue gene. In *Workshop on Massively Parallel Processing (IPDPS'01)*, San Francisco, CA, April 2001.

[32] Gengbin Zheng, Arun Kumar Singla, Joshua Mostkoff Unger, and Laxmikant V. Kalé. A parallel-object programming model for petaflops machines and blue gene/cyclops. In *NSF Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium(IPDPS)*, Fort Lauderdale, FL, April 2002.

[33] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.