Performance and Productivity in Parallel Programming via Processor Virtualization

Laxmikant V. Kalé
kale@cs.uiuc.edu
Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

We have been pursuing a research program aimed at enhancing productivity and performance in parallel computing at the Parallel Programming Laboratory of University of Illinois for the past decade. We summarize the basic approach, and why it has improved (and will further improve) both productivity and performance.

The centerpiece of our approach is a technique called processor virtualization: the program computation is divided into a large number of chunks (called virtual processors), which are mapped to processors by an adaptive, intelligent runtime system. The runtime system also controls communication between virtual processors. This approach makes possible a number of runtime optimizations.

We argue that the following strategies are necessary to improve productivity in parallel programming:

- *Automated resource management via processor virtualization*
- *Modularity via concurrent composability*
- *Reusability via frameworks, libraries, and multi-paradigm interoperability*

Of these, the first two directly benefit from processor virtualization, while the last is indirectly impacted. We describe our research on all these fronts.

1. Introduction

Parallel programming is more difficult than sequential programming because of the additional issues of determinism, synchronization, communication costs, load imbalances and performance portability that must be addressed by the programmer. As a result, productivity of parallel programming efforts tends to be low.

Recognizing the importance of high productivity, in the early days of parallel computing researchers aimed at automatic parallelizing compilers. However, after decades of very stimulating research [7, 37, 17, 18, 9], it has become clear that although some of the tools produced can indeed extract almost all the parallelism from the *given* code, a from-scratch parallel reformulation is often required to attain higher performance.

We have been pursuing an approach to high productivity with scalable performance even for complex, dynamic parallel applications for the past decade [25]. One of the guiding principles for us is to seek an optimal division of labor between the programmer and the "system". The human programmers do what they can do best, while leaving only what can be efficiently automated to the system. Specifically, we find that programmers are best at finding and expressing the natural parallelism of the application, but the runtime system can efficiently carry out resource management and many performance optimizations.

We think that parallel programming productivity can be increased by advancing the state of art on the following fronts:

- **Automatic resource management:** Writing a parallel program involves managing and allocating resources including processors, memories and networks to application data and computations. Especially for irregularly structured and/or dynamically varying applications, such resource management entails a significant programming effort. At the same time, advances in algorithms create smarter algorithms (with lower total operation counts) that tend to be irregular in structure. For example, for N-body interactions, a simple O(n^2) algorithm is easy to parallelize, where O($n \log n$) algorithms (such as Barnes-Hut) or O(n) algorithms (such as Fast Multipole) are more complex. Applications themselves are tackling more dynamically evolving scenarios, typically requiring adaptive refinements as the computation progresses. If the programmer were freed from dealing with resource management issues, their burden would be significantly reduced.

- **Concurrent compositionality:** It should be possible to compose independently developed parallel modules into an application, in such a way that the execution of composed modules may overlap in time or space (i.e. processors); moreover this "concurrent composition" must be achieved without losing efficiency. With this capability, it would be possible for application structure to be based on logical interactions of its modules, automatically overlapping the computation and communications across modules.

- **Techniques for promoting reuse of parallel software components:** Because a parallel module operates in a more complex context, it is more difficult to reuse it than a sequential component. Yet, the complexity of parallel software puts a higher premium on reuse. Thus, we must develop techniques that eliminate the barriers to reuse of parallel software.

In this paper, we illustrate the research we have been carrying out towards these objectives. One of the enabling factors in our research is the idea of processor virtualization. We begin with a brief exposition of this idea.

2. Processor Virtualization

Processor virtualization is a simple idea: the programmer decomposes the computation, without regard to the physical number of processors available, into a large number of objects, which we call **virtual processors (VPs)**. The programmer leaves the assignment of virtual processors to physical processors to the runtime system. The virtual processors themselves can be programmed using any programming paradigm: e.g. they can be MPI "processes" implemented as user-level, extremely lightweight, threads (NOT to be confused with system level threads or Pthreads), that interact with each other via messages, as in Adaptive MPI [20]. Alternatively, they can be organized as indexed collections of C++ objects that interact via asynchronous method invocations, as in Charm++ [28].

This simple idea has significant consequences. Most importantly, from the point of view of this paper, it empowers the **run-time system (RTS)** to optimize resource allocation by migrating VPs across processors. The RTS

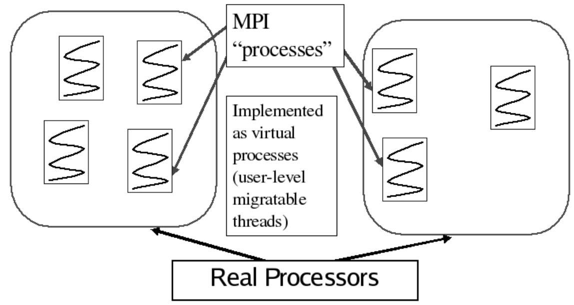


Figure 1. Processor Virtualization in Adaptive MPI: An MPI process is implemented as a user-level thread, several of which can be mapped to one single physical processor.

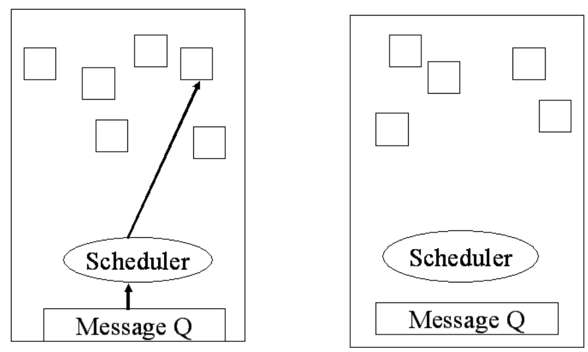


Figure 2. Message-Driven Execution with a processor-level scheduler

is also involved in delivery of messages to VPs. as a result, it can optimize communication as well.

Let us first sketch the direct consequences of processor virtualization: since each physical processor may house hundreds (or even thousands) of VPs, the RTS needs to have a scheduler to decide which VP executes next. This scheduler can (and indeed must) be message-driven: it only schedules VPs that are ready to execute because they have a message pending. This message-driven scheduler turns out to be a critical component from the point of view of concurrent composition.

Second, since VPs may migrate as a program evolves, the RTS needs to maintain information about where each VP is located. This can (and must) be done efficiently, without bottlenecks. Our implementations ensure that in most cases, messages are delivered to VPs without any forwarding, with the assumption that migrations are not as common as messages [33].

Charm++ and Adaptive MPI are systems we have developed over the past 14 years that embody this idea of processor virtualization. For concreteness, the next few sections assume each VP as running an MPI "process" and interacting with others via the usual MPI sends and receives.

3. Automated Resource Management

Processor virtualization empowers the run-time system (RTS) to incorporate intelligent optimization strategies. We discuss two categories of such strategies below.

3.1. Automatic Load Balancing

Probably the most obvious advantage of processor virtualization is that the runtime system can do automatic load balancing dynamically. Since the application program never sends messages directly to physical processors, the RTS is free to migrate the VPs across processors any time it pleases.

Of course, the RTS must be quite "intelligent" for this to work, but it is certainly possible [33]; what is more, only one RTS needs to have these smarts, whereas all the application programs can just use it.

In the simplest possible setting, the RTS can monitor the load on a processor and its neighbors. If/when a (physical) processor goes idle, the RTS sends a request for additional VPs from a neighboring processor. Other variations on this idea are possible [23, 39].

A more interesting and fruitful set of strategies becomes feasible when we observe a property of many parallel computations, especially those involving physical systems. Even for dynamic applications, the computation loads and the communication patterns exhibited by the VPs tend to persist over time for most of the VPs. This is because often dynamic variations happen abruptly but infrequently (as with periodic mesh refinements) or frequently but slowly (as with migrations of particles in n-body codes including molecular dynamics).

Based on this "principle of persistence" (which is a heuristic principle, like the principle of locality), one can now build measurement based runtime load balancing strategies. The RTS can instrument the VPs to record computational load and communication patterns. It can do this automatically, without user code, since the RTS is the intermediary for both scheduling and communication. Load balancing strategies can then use this database in a centralized or distributed manner to effect remapping decisions periodically. (These periodic decisions can be augmented by "idleness-based" schemes as mentioned above when necessary.)

We and others have implemented many such strategies, and work is ongoing on strategies that observe more subtle patterns, such as dependences, critical paths, multiple-phases-within-iterations and so on. However, the main point is that the application programmer doesn't have to worry about this important aspect of their parallel program.

As a concrete example, we show dynamic load balancing in action in NAMD, the highly scalable molecular dynamics program used routinely by biophysicists. Figure 3 shows processor utilization against time for a NAMD run of 1024 processors [29]. The initial greedy balancer works from 157 through 160 seconds (the period in the graph with the dip in utilization), leading to some increase in average utilization. Further, after the refinement strategy finishes (within about .7 seconds) at around 161.6 seconds, we can see that utilization is significantly improved. In this figure, we may appear to spend too much time on load balancing; however, in molecular dynamics, such load balancing is needed only after several thousand timesteps [38].

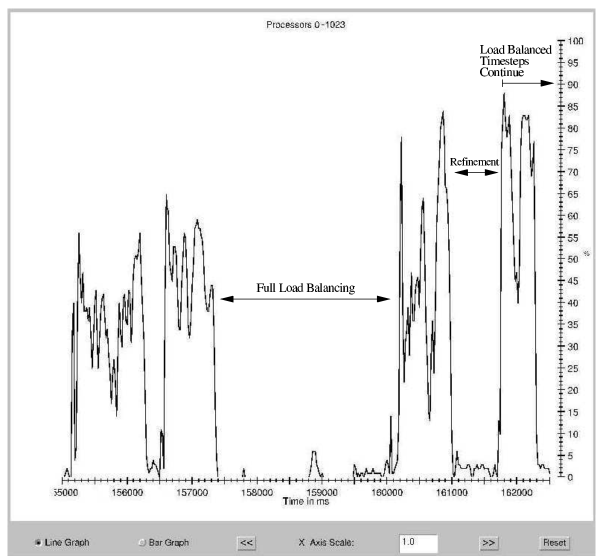


Figure 3. Processor Utilization against Time on 1024 processors

As a result of such runtime optimizations, NAMD has attained an unprecedented high performance on several thousand processors, leading to a Gordon Bell award [38]. The performance of NAMD using Charm++ on PSC Lemieux is shown in Figure 4. Each timestep takes around 25 seconds on 1 processor. This drops to 27 milliseconds on 1000 processors and finally down to 12 milliseconds when scaled to 3000 processors with a corresponding performance level of 1000 GFLOPS. Not only are the achieved speedups impressive, the absolute time taken per timestep (12ms) is also lowest by a considerable margin compared with other molecular dynamics programs.

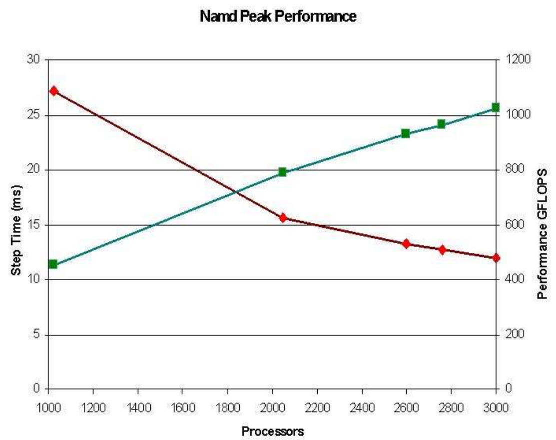


Figure 4. Processor Utilization against Time on 1024 processors

3.2. Communication Optimizations

Since the RTS mediates communication, it can intercept the communication and replace communication algorithms as required by the patterns observed. This is especially true for collective operations. By keeping track of the number of processors (VPs and physical) involved, the amount of data, and the state of the rest of the computation, the RTS can decide which of the available collective communication algorithms will be better suited, and switch it at runtime. Our own results in this area have been promising [**?**].

Automatic runtime communication optimizations can also be performed for other non-collective operations. For example, a graph partitioning application written by a newcomer to parallel programming used an extremely fine-grained communication style: very short messages with a few bytes of data were being sent, which would have led to bad performance. However, by interposing the streaming library of the communication subsystem of the RTS (which collects short messages locally, and sends them using a virtual mesh), this was optimized without changing user code. In this case, the interposing was done manually, but as the RTS capabilities are improved it can make such decisions itself. (Even with the manual interposing, the advantage still remains that the user code didn't have to change.)

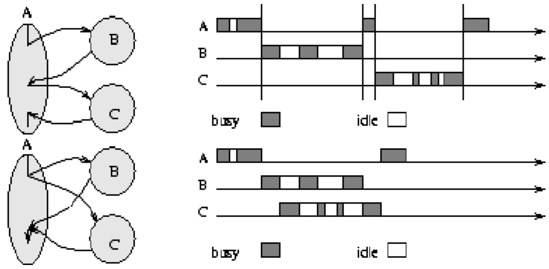
4. Modularity via Concurrent Composition

For high productivity in parallel programming, one should be able to modularize the program. In particular, it should be possible to compose independently developed parallel modules into a single parallel application (or into higher level modules, composed hierarchically). Further, the modules being composed should be allowed to overlap their execution in time, and over processors. Without this flexibility, one risks the danger of fragmenting the set of processors (especially when a large number of modules are being composed) and certainly loses the ability to exploit adaptive overlap of communication and computation across modules. This is illustrated with a schematic and application example below.

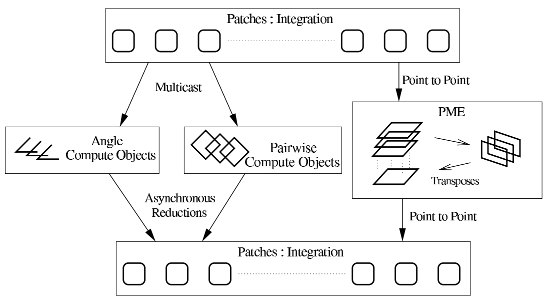
Consider the situation in Figure 5(a). A, B and C are each parallel modules spread across all processors. A must call B and C, but there is no dependence between B and C. In traditional MPI style programming, one must choose one of the modules (say B) to call first, on all the processors. The module 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 dependence between the 2 modules.

In contrast, with processor virtualization (and the message-driven execution induced by it), A invokes B on each 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 availability 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 processor virtualization, the code in B does not have to know about the code in A or C, and vice versa.

This phenomenon is illustrated in NAMD (Figure 5(b)). The computation partitions atoms into a set of cubic cells called "patches". Interactions between atoms in adjacent cells are computed by separate virtual processors called the "pairwise compute objects" in the Fgure. The PME (Particle-Mesh Ewald) module involves two 3D-FFTs (each with a communication intensive transpose operation) over a relatively small grid (192x144x144 in one case). By concurrently composing the PME and force-calculation modules, it becomes possible to use the considerable latency of the transposes in the PME algorithm with pairwise-force computations adaptively. Neither partitioning of processors among the two modules, nor sequencing their execution one after



(a) Modularity and Adaptive Overlapping: Schematic



(b) Concurrent Composition of PME and Force Computations in NAMD

Figure 5. Concurrent Composition

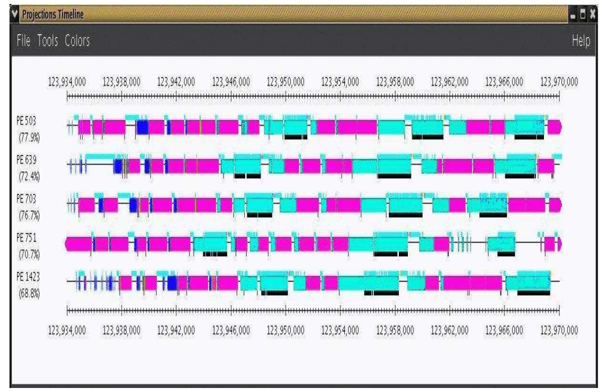


Figure 6. PME Execution in NAMD

the other will yield the same efficiency of concurrent composition employed by NAMD. Moreover, this efficiency is attained without any coding by the programmer to juggle execution between the two modules.

Figure 6 shows the timeline of a few processors in a 2112 processor NAMD run on PSC's Lemieux alpha-cluster. The light gray rectangles (as well as the darkest gray rectangles at the beginning, around 123.938 secs) represent components of the PME computations, whereas the medium gray rectangles are pairwise (and bonded) force computations. The overlap of the two modules' operations can be clearly seen. On PSC's Quadrics communication network, the communication co-processors ensure that the CPU spends only a small time on communication. Therefore, all the latency of the transpose operation (between the yellow sections) is available for doing useful pair-wise force computations, which are adaptively scheduled by the system.

4.1. Software Engineering Benefits

Virtual processors are logical entities, and can be made to correspond to the structure of the application. In contrast, shoe-horning the application structure into physical processors leads to inelegant software.

The simplest example of this is in the number of processors used. MPI programs modeling a physical domain via structured grids often require the number of processors to be a cube (and even a power-of-two cube). With virtualization, one can decompose the data into a power-of-two cube virtual processors, yet be able to use the available number of physical processors. Attempting to do that without explicit support for virtualization leads to multiple-block codes that have to deal with messages for different blocks at various points in the code, and can spoil neat expression of the evolution of a single block when it engages in multiple phases of communication.

In software engineering terminology, such parallel software (based on physical processors) often lacks "cohesiveness". Code and data are brought together simply because they are on the same physical processor.

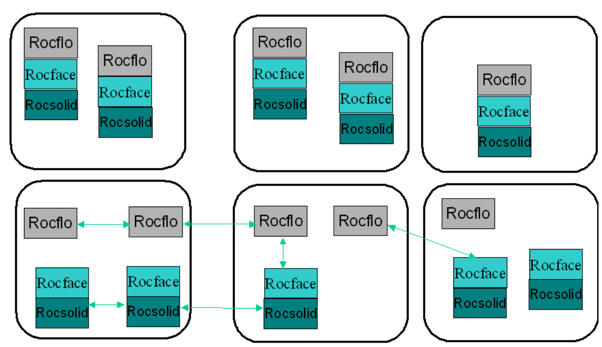


Figure 7. Rocket simulation via virtual processors

Consider a version of the rocket simulation applicationconsisting of two parallel modules: Rocflo (a 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 the fluids and solids meshes were decomposed separately 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 top). An AMPI implementation, on the other hand, (Figure 7 bottom) 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).

5. Cost of Processor Virtualization

An important question often raised is about the cost of processor virtualization. Although users may be willing to concede a certain amount of performance in return for benefit in productivity, they would like to know the extent of the performance loss. The situation in reminiscent of early days of (Fortran) compilers, when users were unwilling to switch away from assembly language programming. In fact, then as now, since programmers are highly conscious of performance issues, and they already have (by compulsion) mastered the intricacies of low level programming, they will not want to switch to a new paradigm unless assured of “as good” performance with lower effort.

To be sure, the adaptive runtime systems enabled by processor virtualization achieve such performance enhancements as automatic dynamic load balancing. But it can be argued that expert programmers will be able to achieve such performance by programming load balancing code into their application themselves.

So, the question of overhead is still important. Luckily, in most situations, the overhead of processor virtualization is insignificant. Context switching between virtual processors requires less than a microsecond on current processors. (Recall that our virtual processors are not system level threads, or pthreads; they are user level threads). The number of messages increases with multiple VPs per processor. Messages have a software overhead of a few microseconds. So, the degree of virtualization chosen should be such that the computation per message is substantially larger than these overheads. This is clearly reasonable for most applications. For example, in a particular molecular dynamics benchmark, we used about 30,000 VPs spread over 3000 processors; the average computation per VP per timestep was 900 microseconds, and the average computation per message was about 200 microseconds.

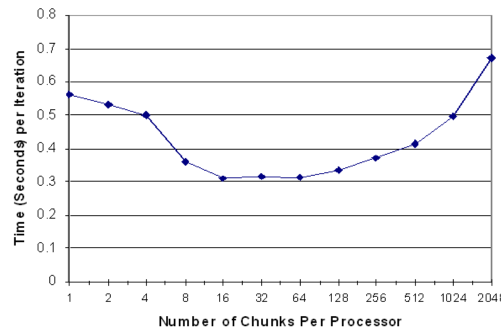


Figure 8. “Overhead” of Multipartitioning in an FEM application

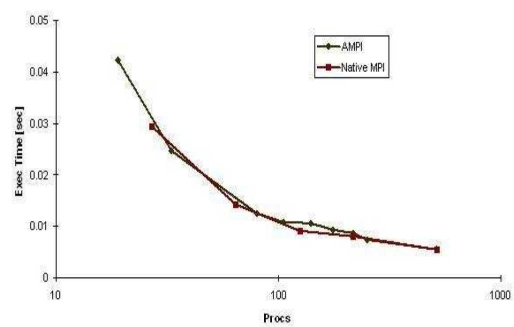


Figure 9. 7-point stencil on a 3D problem size 240^3 run on PSC Lemieux.

To compare the performance of MPI and AMPI, we compared the performance of a 7-point stencil code, doing Jacobi relaxation for 3-D data, in Figure 9. AMPI achieves nearly identical performance as MPI, but it runs on any number of processors, whereas MPI requires a cubic number of processors. Additional performance data can be found in [20].

Cache performance typically improves with processor virtualization, because of its blocking effect. A study of the effect of virtualization we did with an unstructured mesh application showed (Figure 8) that performance actually improves with the degree of virtualization, and only after over 1024 VPs per physical processor does the overhead start showing its effect.

Since Charm++/AMPI are often implemented on top of native MPI, the communication costs can be expected to be higher. This is not a fundamental cost: on many machines,our implementation uses lower level communication APIs (E.g. Elan, GM and VMI), where our performance is comparable to MPI [**?**]. A comparison of MPI and AMPI versions of the rocket simulation code at Illinois also showed the performance of two versions to be almost identical [11].

In some applications, virtualization leads to a large number of small messages. This can be mitigated by using a streaming library available in the Charm++ runtime that uses message combining to optimize performance.

5.1. Limitations and Remedies

There are situations where processor virtualization may lead to poorer performance.

When large layers of ghost cells (instead of the common 1-layer ghosts) are used, virtualization may be constrained by the memory overhead of the extra ghost cells. However, we believe it is possible to alleviate this overhead by a combination of techniques alluded to in [26].

Parallel algorithms whose costs increase with the number of processors can also limit the benefits of virtualization. Fortunately, such algorithms are rare, and they exist as components of large applications (e.g. parallel prefix, which has a complexity of $2n + \log p$), and they can be allowed to run with a lower virtualization factor by concentrating data in p virtual processors.

Another source of overhead arises when processors use a large amount of remote data. If each VP uses its own memory for such data, *and if* such data overlaps significantly (i.e. multiple VPs request the same data) then both memory and communication overheads may increase. This happens, for example, in gravity computations, where each cell containing a bunch of particles is a VP, and the computation requests particles from other cells. This can be remedied by using an abstraction for requesting and caching remote data, which is implemented by a lower level library that is aware of physical processors. We are using such a library in a collaborative project in computational astronomy. NAMD also uses a similar technique in the form of proxy objects [24].

6. Reuse of Parallel Software Components

Reuse of parallel components can be promoted by domain-specific frameworks and allowing composition of modules written in different parallel programming paradigms.

6.1. Frameworks and Libraries

One method for improving productivity is to reuse a collection of techniques that are commonly needed in a particular application domain. Even though parallel applications are diverse, one can find such commonalities. For example in simulations of physical models (which constitute the dominant use of parallel computers now), one finds that only a few distinct parallel data structures are used: structured grids (arrays), unstructured meshes, spatially decomposed particles, tree structures (e.g. in multi-grid and AMR), along with a collection of linear system solvers, cover a large fraction of the application space. To improve productivity we should therefore extract the domain specific techniques into frameworks so that they don't have to be recoded for every application.

One can take two approaches to design domain specific frameworks: Vertical integration or horizontal layering. Vertical integration leads to highly specialized problem solving environments (e.g. for structural dynamics), while horizontal layering leads to a collection of capabilities that can be composed in different ways for different applications.

Our experience with horizontally layered frameworks has been quite positive. Specifically, we have developed an unstructured-mesh framework [4] that can partition a mesh and set up communication lists for a user-specified layer of ghosts. Although originally used for Finite Element computations, the framework is now used for finite volume, discontinuous Galerkin, as well as space-time meshes. Other capabilities such as collision detection or matrix-free solver interfaces are available as separate components.

However, the simplicity of problem solving environments (PSEs) suggests that horizontally layered components should be used to put together specialized PSEs, which will cut down on the development cost of PSEs themselves.

It is tempting for application developers to decide to code such capabilities themselves, since they seem relatively simple. However, once one takes the maintenance cost of software into account, and considers the fact that many focused optimizations and capabilities may have been implemented by the framework developer in the context of real applications, the advantage of frameworks becomes clear. Also, frameworks can make use of complex and tricky features of the RTS that application developers may require a significant effort to use.

6.2. Multiparadigm Interoperability

Although reuse of parallel components is desirable for enhancing productivity, another obstacle to such reuse occurs because of the use of different parallel programming paradigms in different modules. One module may be written using Charm++, while another might use a DSM system, or Global Arrays [36], or the FEM framework, or MPI, or BSP, etc. By requiring that all the modules being composed into an application use the same paradigm, we give up a large opportunity for reuse.

The modules themselves may be written in different paradigms for two distinct reasons: First, a particular paradigm may be better suited for the algorithms being specified by the module. Second, it may be simply a matter of subjective choice of the programmer of that module (which might have been developed independently at an earlier time).

Concurrent composition capabilities of message-driven execution come in handy in this context, with one proviso: If all the (message-driven) paradigms share a single scheduler (See Figure 2), and possibly a common runtime support layer, then such interoperability is possible. With this in mind, we designed the Converse [27] framework, which provides (a) common capabilities such as a scheduler, user-level thread package, portable low-level communication interface, and encapsulation of other machine capabilities and (b) methods to allow the concurrent interoperation of modules written in different paradigms. In addition to providing interoperability, Converse also simplifies the task of writing runtime systems for new parallel programming paradigms.

Converse and Charm++ together now support a wide variety of programming paradigms in our infrastructure, including ARMCI, Global Arrays, Adaptive MPI, Jade (a parallel Java-like language), PVM, specific forms of DSM systems, etc. Of course, adoption of such multi-paradigm frameworks is possible only when runtime developers agree to a common standard, for which Converse is but one candidate.

7 Related Work

This paper focused on productivity-and-performance oriented ideas developed by the author, for pedagogical clarity. However, many of the central ideas have appeared in other research as well.

Chare Kernel, the C-based progenitor of Charm++ was developed around 1989 [22]. This system, with function calls to remote processors with objects encoded as global pointers, and a message-drive scheduler, is similar to Nexus [14]. Active Messages [41] shared message-driven execution ideas with Charm, but not processor virtualization. The Actors model [1], with its message-driven objects, is quite similar to Charm++ at its lower level, and is considered useful for specifying and understanding the semantics of message-driven programs. However, the intellectual progenitors of our early work were the RediFlow project [31] for parallel execution of functional programs, and the Dataflow research. The basic low-level ideas in Charm++ can be considered to be macro data-flow, extended with high-level notions of automatic resource management. Other research with overlapping approaches include work on Percolation and Earth multi-threading system [21], work on HTMT and Gilgamesh projects [15], and the work on Diva [16].

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 [35] is an example of an approach based on virtualization that is closer to our work. 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.

In the direction of interoperability, recent work on Common Component Architecture [2] is important, as it provides a method for interconnecting independently developed modules, enhancing reuse. We believe that it needs to be extended to allow processor virtualization, which is infeasible in the current form.

Several other domain specific frameworks exist that aim to raise the level of abstraction in programming. For structured grids, with possible adaptive mesh refinements, they include KeLP [13], Paramesh [34], and Chombo [8]. For computations on unstructured meshes, frameworks such as Sierra [40] exist.

Since linear system solvers arise in many current parallel applications, several libraries provide good support for them such as ESI [32] and PETSc [3]. Numerical libraries such as Ellpack [19] and Linpack [12] also help enhance reuse.

Shared memory programming models, and especially with its standardization via OpenMP, must also be considered. Via systems such as TreadMarks [30], such models are now available on distributed memory machines. However, the claim that shared memory abstraction simplifies parallel programming has not quite been substantiated. Although some programs look simpler with shared memory, others get more complex, especially if they have to deal with race conditions. It is possible that the full generality of a shared variable is unnecessary, while limited use of shared variables, in specific modes, might be productive (E.g. GA [36]).

8. Productivity Metrics

We have not performed any quantitative studies of improvement in productivity with Charm++/AMPI yet. We will state some anecdotal evidence instead.

Clearly, when an application requires dynamic resource management, the savings in writing code are apparent. For example, in multi-block codes, one has to write by hand how the blocks should be distributed after new refinements. All that code is in the Charm++ runtime system, and is being reused.

Virtualization also confers benefits by reusing the ability to migrate objects in different contexts. For example, once one has written a Charm++ or AMPI program with migratable objects, the runtime system can automatically carry out efficient check-pointing, support out-of-core execution, change the set of processors (shrink or expand) used by the application at runtime, vacate a machine that is about to go down or needs to be relinquished to the owner, and support fault tolerance. All of these new functionalities can be available without the user having to write new code. Of these, fault tolerance is still being worked on. Once the runtime implements this feature, it will be available for all applications without significant new application code.

Further productivity enhancements are expected when we are able to develop a "standard library for parallel programming" which will eliminate having to write code for commonly needed parallel operations.

9. Conclusion and Future Work

We presented a research agenda, and our progress along it, which has been explicitly aimed at improving programmer productivity and computer performance on complex parallel applications.

Processor virtualization was seen as a key to some productivity enhancements. Via this, the runtime system is empowered to carry out intelligent optimizations, including dynamic load balancing and communication optimizations, without programmer intervention. It also leads to message-driven execution, and thus to the ability to concurrently compose multiple independently developed modules effectively without losing efficiency. Separation of virtual processors from physical resources in the programmer's mind also leads to a separation of concerns and better software engineering practices.

It can be argued that manual, application specific resource management can always do at least as well as automated techniques, in terms of performance. However, with increasing complexity of applications, and the increasing number of processors in large supercomputers, we believe that automated techniques, culled from experience on a wide variety of applications, will be more efficient than what most actual programmers accomplish by themselves, even with a lot of effort.

Concurrent composition enables flexible reuse of parallel modules. But to make such reuse happen, reusable parallel modules must be developed. Based on the idea that a relatively small number of basic data-structures account for a large number of application components, we advocate the building of domain-specific frameworks. If each such framework provides encapsulation of a limited but useful capability, it becomes possible to compose such frameworks into vertically integrated problem solving environments.

Modules written in different parallel programming paradigms can be integrated if they share common runtime structures, and especially a message-driven scheduler (in case of virtualized or user-level thread based systems). We described our experience with Converse, an infrastructure explicitly designed to support interoperability and easy development of runtime systems for new programming paradigms.

In this paper, we kept the focus on our research in order to present a single point of view. There are many other approaches aimed at productivity. Interaction and cross-fertilization of ideas among them will lead to better systems/approaches for the future. As an example, we hope that the common component architecture effort can be extended to permit virtual-processor based formulations.

We have identified some additional future directions towards productivity. The research on intelligent adaptive runtime systems, although quite fruitful, has only picked the "low-hanging fruit". We see potential for much more sophisticated runtime techniques, based on self-observing systems. In terms of low-level support, co-processors that can handle remote requests (beyond just puts and gets) are essential for effective deployment of composable systems. At the higher end, it seems possible to include compile-time support in a comprehensive approach aimed at productivity. The current obstacles for this include the fact the compiler-support issues that arise in this context are often considered mundane, and are not the usual issues (such as automatic parallelization) that are considered attractive by the compiler community. Telescoping languages being proposed and developed by several researchers might be able to bridge this gap.

We also see potential to increase productivity via additional language support. For instance, Jade [10], a language based on Java, provides the ability to take advantage of some of the simplifications in Java, such as the use of references instead of programmer managed storage, while still having access to the features of Charm++ and the Converse runtime.

Also, we see potential in the ability to build systems of composable parallel components. The Charisma system [6] is a start in this direction, with the concept of explicit runtime support for components. A future orchestration language, which will allow the interactions among components to be defined in a scripting language, will also improve the reuse of parallel components and make the logic of parallel applications more explicit.

10. Acknowledgments

The research alluded to in this paper would not have been possible without generations of graduate students. For specific help in preparing this paper, I am grateful to Gengbin Zheng, Sameer Kumar, Chee Wai Lee, Chao Huang, Mark Hills and Orion Lawlor. The paper builds upon an overview of processor virtualization presented at LACSI 2002 [26]. The work which is summarized in this paper has been supported by grants from NSF, NIH and DOE over the years, including recent grants (NSF NGS 0103645, NSF ITR 0121357, NIH PHS 5 P41 RR05969-04, DOE ASCI B341494).

References

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

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, pages 115–124, Redondo Beach, California, August 1999.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.1, 2001.

[4] M. 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.

[5] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. 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.

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

[7] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. 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.

[8] Chombo – infrastructure for adaptive mesh refinement. http://seesar.lbl.gov/anag/chombo/.

[9] R. Cytron, D. J. Kuck, and A. V. Veidenbaum. The effect of restructuring compilers on program performance for high-speed computers. *Computer Physics Communications*, 37(1–3):39–48, 1985.

[10] J. DeSouza and L. V. Kalé. Jade: A parallel message-driven java. In *Proc. Workshop on Java in Computational Science, held in conjunction with the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia and Saint Petersburg, Russian Federation, June 2003.

[11] E. deStruler, J. Hoeflinger, L. V. Kale, and M. Bhandarkar. A New Approach to Software Integration Frameworks for Multi-physics Simulation Codes. In *Proceedings of IFIP TC2/WG2.5 Working Conference on Architecture of Scientific Software, Ottawa, Canada*, pages 87–104, October 2000.

[12] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM Press, 1979.

[13] S. J. Fink, S. B. Baden, and S. R. Kohn. Flexible communication mechanisms for dynamic structured applications. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 203–215, 1996.

[14] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, May 1994.

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

[16] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin. Mapping irregular applications to DIVA, A PIM-based data-intensive architecture. pages ??–??, 1999.

[17] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine independent parallel programming in fortran-d. In J. Salz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier Science Publishers B.V., 1992.

[18] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for fortran-d on mimd distributed memory machines. In *Proceedings of Supercomputing 1991*, Nov. 1991.

[19] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, K. Y. Wang, and S. Weerawarana. //ellpack: a numerical simulation programming environment for parallel mimd machines. In *Proceedings of the 4th international conference on Supercomputing*, pages 96–107. ACM Press, 1990.

[20] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

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

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

[23] L. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Computer Science*, volume 748, pages 12–41. Springer-Verlag, 1993.

[24] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[25] 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.

[26] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[27] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[28] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[29] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop (ICCS'03)*, Melbourne, Australia, June 2003.

[30] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

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

[32] M. G. Knepley and et al. Solvers as operators - proposal for the esi solver interface.

[33] O. S. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[34] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, 2000.

[35] 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.

[36] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. In *Journal of Supercomputing*, volume 10, pages 169–189, 1996.

[37] D. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):829–842, dec 1986.

[38] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[39] V. A. Saletore. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. In *Proceedings of the Fifth Distributed Memory Computing Conference (5th DMCC'90)*, volume II, Architecture Software Tools, and Other General Issues, pages 994–999, Charleston, SC, Apr. 1990. IEEE.

[40] 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.

[41] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.