Parallel Objects: Virtualization and in-Process Components

Laxmikant Kale Orion Lawlor Milind Bhandarkar
Department of Computer Science
University of Illinois at Urbana-Champaign
kale@cs.uiuc.edu, olawlor@acm.org, milind@cs.uiuc.edu

Abstract

We summarize the object-based virtualization model that we have been developing for the past decade, and demonstrate how it enables automatic optimizations, especially at runtime. The parallel programming paradigm represented by the virtualization model has been implemented in the Charm++ and AMPI libraries. In this paradigm, the programmer specifies their parallel application as a collection of interacting entities, without any reference to processors. The runtime system is free to map these entities to processors, and migrate them at runtime as needed. This separation of concerns enables several runtime optimizations, involving message-driven execution, automatic load balancing and communication patterns. A recently developed component model is also shown to create new opportunities for runtime optimizations.

1 Introduction

Developing complex parallel applications that run efficiently on large parallel machines is difficult because the programmer has to handle many interrelated issues. Two issues that are of interest here are the specification of the parallel algorithm, and its efficient parallel implementation. Most current parallel programming models require the programmer to specify both of these together, leading to complex, unwieldy programs. For example, it is not enough to specify parallel loops in a shared memory program; one also has to privatize variables, split locks, make artificial loop transforms, etc. to improve efficiency. In MPI, in addition to specifying the algorithm in terms of processes and when and what data they communicate, the programmer is forced to also use different variants of send and receive calls, and move sends and receives up and down in the program, to improve efficiency. Further, even when a natural decomposition for a parallel algorithm is available (such as a spatial grid of cells, or oct-trees) an MPI programmer is typically required to break the problem into one piece for each available processor, leading to awkward decomposition schemes, and/or unnecessary restrictions on the number of usable processors (e.g., a cube, or a power of two, or both).

The methodology we have been been pursuing for the past decade [4, 6, 3, 11] is to separate decomposition (parallelization) from assignment of work. Decomposition cuts the work to be done into parallel parts, called chunks, objects, entities, threads, virtual processors, and so on. As-

signment is the mapping of work to processors, and the sequencing of work on each processor.

We require that the programmer only specify the problem decomposition, while having the runtime system automatically manage work assignment. In this model, the programmer specifies the communication only by naming the chunks, and not the processors. This gives the runtime system the freedom to move the chunks among processors as it sees fit.

This approach leads to an efficient division of labor between the system and the programmer. It also creates opportunities for automatically optimizing performance, especially at runtime. In this paper, we will describe what these opportunities are, and what we have done to harness them. Several production quality applications have been developed using this approach, with unprecedented speedups achieved in many cases. These applications will be used to illustrate the use of our runtime optimizations.

In the next section, we describe the model further, and two of its realizations, in Charm++ and AMPI. Section 3 identifies optimization opportunities created by the model and demonstrates optimization techniques developed and in development.

How can we go to a higher level programming paradigm from the virtualization model? We have been exploring two orthogonal approaches: parallel components and domain-specific frameworks. They both increase the degree of reuse of parallel software modules in their own ways. Section 4 describes the component model that is facilitated by the virtualization paradigm, and the optimizations enabled by it. With this, each parallel component module can be reused in a variety of contexts, unchanged, because the code for "connecting" components together is taken out of the component itself. The second approach of domain-specific non-intrusive parallel frameworks is summarized in section 5. The conclusion forms section 6.

2 Virtualization

In the following sections, we present two concrete implementations of the virtualization idea: Charm++ and AMPI.

2.1 Charm++

Charm++[4, 5, 9] is an object-oriented parallel language that provides remote method invocation for C++. Charm++'s basic, programmer-visible unit of computation is a C++ class, not a processor; this is a significant difference from MPI. Charm++ employs the virtualization concepts presented in this paper by allowing a processor to host many independent parallel objects.

The Charm++ execution model is message-driven—that is, computations are triggered by message arrivals. A very simple scheduler picks the next available message and uses it to invoke a method on the appropriate object. From an object's point of view, an incoming remotely-triggered method invocation looks exactly the same as an ordinary local method invocation.

Outgoing messages are sent via "communications proxy" objects. To send a message to an instance of a user-defined class that lives on another processor, one invokes a method on a small "proxy" instance. The proxy's methods simply package up the parameters and send them off to the real object. Charm++ generates the C++ code for a proxy class automatically, based on a description of the real class's remotely accessible methods and parameters.

Thus far, Charm++ is quite similar to CORBA or Java RMI. However, Charm++'s remote method invocation is asynchronous, making "send a message" and "invoke a method" essentially equivalent operations. In addition, Charm++ aims for high performance, and typically only imposes a few microseconds of overhead on native MPI communication.

In Charm++, a parallel object can also migrate from one processor to another under either direct application or run time system control, allowing automatic, dynamic, application-independent load balancing as described in Section 3.5. Parallel objects still receive messages properly and participate in collective communication even when migrations are occurring[11].

Charm++ has been used as a foundation for several real applications, including the scalable molecular dynamics program NAMD[12] and the Charm++ FEM Framework[1].

2.2 AMPI

Adaptive MPI, or AMPI[3], is a virtualized implementation of MPI in which several MPI processes can efficiently share a single processor. This virtualization is achieved by making each traditional MPI process a Charm++ migratable "user-level" thread, which can be thought of as a virtual processor. Unlike traditional threads, which are created and switched by the operation system kernel, user-level threads are created and switched by ordinary user code, which makes them very efficient—the user-level thread context switch time on modern machines is under a microsecond. Since Charm++'s user-level threads are migratable, AMPI processes too be migrated like any other Charm++ object. Because there are several threads per processor, AMPI enjoys all the benefits of virtualization.

MPI programs that use global variables in a non-threadsafe manner cannot immediately be run under AMPI. The global variables must be privatized, either manually or via a special source-to-source translator.

Because the processes of an AMPI program are virtualized, AMPI can also be used to allow several different MPI programs to efficiently share a single physical processor. Different MPI programs can communicate using a generalization of the MPI_COMM_WORLD communicator called MPI_COMM_UNIVERSE. AMPI also uses the same virtualization facilities used for migration to allow checkpointing, which is seen as a kind of migration to disk.

3 Virtualization Optimizations

3.1 Message Driven Execution

Message-driven execution, the parallel counterpart to data-driven execution, is based on the very simple, almost tautological idea that a processor should work on the currently available data. Because the programmer cannot and should not know the exact order in which messages will arrive at a processor, the program cannot and should not have the exact sequence of message arrivals encoded in it. Instead, the runtime system maintains a very simple scheduler that reads each incoming message from the network, and passes the message on to the appropriate part of the program.

For a toy example, a small sequence of exact MPI source/tag match receive calls can be made message-driven by replacing them with a "wildcard receive" followed by a switch statement. By not imposing an artificial order on the incoming messages, we can process the messages in the order they arrive, maximizing efficiency.

But for a complex, multi-module program written by independent developers, many different kinds of messages might arrive, so the switch statement following a wildcard receive can become an unmaintainable mess. For maintainability, some sort of abstraction, like tag registration on program startup, must be imposed. Thus a pursuit of efficiency (processing messages in the order they arrive) and maintainability (avoiding one giant switch statement) in parallel programming inevitably leads to a full-fledged message-driven execution support system. Like the eye in evolutionary biology, the idea of message-driven execution has likely independently evolved from scratch several times.

3.1.1 Promoting Modular Design

Consider two parallel libraries, A and B, each of which run on all the available processors. If A makes a blocking receive call while running, a common approach in MPI, then B will be unable to use the processor. But if B does not depend on A, this is wasteful– in a message-driven system, B can run while A is waiting for data, and vice versa. As shown in Figure 1, interleaved execution can be much more efficient than ordinary, blocking execution.

Without a message-driven system, it is still possible to achieve interleaved execution, but only if the programmers responsible for A and B were aware of the problem and manually inserted calls to each other's libraries. But this solution introduces unnecessary coupling between A and B, prevents the operation of A without B and vice versa, precludes the addition of some new library C, and still does not respond to the actual message delivery at runtime. By contrast, a message-driven system naturally, automatically interleaves any number of components in a responsive manner.

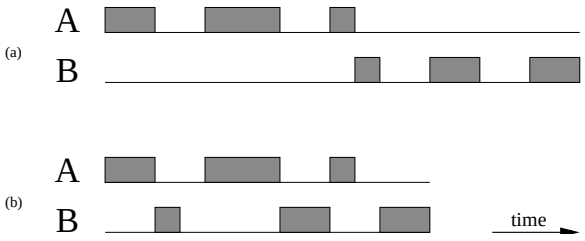

Figure 1: If two parallel libraries, A and B, execute one after the other, as in (a), the libraries idle times are not hidden. Interleaved execution, as in (b), is more efficient and happens automatically in a data-driven system.

3.1.2 Communication/Computation Overlap

In the previous section, we saw how message-driven execution allowed two parallel libraries A and B to alternate their use of the CPU, improving utilization and hiding network latency. The same overlapping occurs anytime several parallel objects share a single processor—even if the all the objects are of the same type. This is one reason why we encourage the use of more parallel objects than processors.

Another major benefit of having several parallel objects per processor is a form of communication pipelining that occurs. Figure 2 shows how splitting a computation into several pieces decreases the apparent network latency and improves overall performance.

3.2 Out-of-core execution

Parallel applications often have large memory requirements. In cases where the memory required to model the particular application is larger than

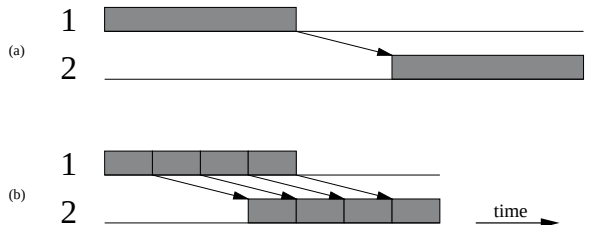

Figure 2: Coarse-grain computations lead to large communication latencies, as shown in the two-processor utilization diagram in part (a). Fine-grain computations, as in (b), allow communication to be pipelined, increasing processor utilization and decreasing the impact of network latency on performance.

the available memory, traditional virtual memory works by swapping parts of the application to disk. Because swapping is done on a page-by-page basis, and because there is no way to predict the next page that will be needed, traditional virtual memory has a substantial performance cost. Prefetching is frequently used to hide latency and hence improve the performance of memory systems. Virtualized parallel programs provide an excellent opportunity to perform intelligent prefetching, because in this context some knowledge of the future *is* actually available.

In a data-driven system, whenever a message arrives, it is inserted in the scheduler's queue. The scheduler retrieves messages from the queue and processes them. The scheduler's queue, then, essentially lists the objects that are about to run. Prefetching based on the message queue has proven to dramatically improve performance[15].

The actual prefetching can be preformed by having a special "victim" thread touch the memory of each soon-to-be-executed object, taking a page fault if the object is not in memory; the ordinary worker threads then never experience page faults. Alternatively, objects can be explicitly written to and read from disk, which avoids any dependence on operating system virtual memory.

3.3 Automatic Checkpointing

Checkpointing a Charm++ program involves saving the state of all of the programs' objects to disk, which can be viewed as migrating the objects to disk instead of to another processor. That is, checkpointing can be implemented as a special case of migration, and the same migration support can be used to serialize an object to disk.

If some of the processors in the system fail while the program is executing, the program can resume execution from its last checkpoint on the processors in the system that are still working. Objects that were on a failed processor will resume their execution on another functioning processor. A prototype checkpointing facility was implemented for Charm++ [13], this work included the ability to restore a checkpoint on a different machine architecture.

3.4 Principle of Persistence

Parallel applications written using the Charm model often exhibit the *principle of persistence*: the communication patterns and computational loads of the objects tend to persist over time. This can be thought of as the parallel analog of the *principle of locality*, the heuristic on which the memory hierarchy is based. The principle of persistence holds in most adaptive and dynamic applications as well, because changes in these patterns either (a) are small, slow and

continuous (as in molecular dynamics), or (b) large, but infrequent (as in adaptive refinements in AMR). When the principle holds, several runtime optimizations are enabled, such as measurement based load balancing, and communication libraries that learn and adapt to evolving communication patterns. Some of the subsections below illustrate the exploitation of this principle.

3.5 Object Migration and Automatic Load Balancing

Charm++ and AMPI support object and thread migration, which can be used for a number of purposes. One of the most important is load balancing—to decrease the amount of work a processor has to do, we need only migrate a few of its objects away. Because processors are programmer-transparent, work migration does not affect the running computation.

Charm++ includes a load balancing framework[8], which monitors the compute load and communication patterns for each running object. A pluggable "strategy" module uses this information to decide which objects to migrate, and where. Different strategies take into account different factors—some attempt to equalize the compute load, some minimize the number of migrations, some optimize object communication, and some do all three.

In the following sections, we describe some other uses of object migration.

3.5.1 Flexible use of desktop machines

Desktop workstations present a large, largely untapped parallel computer available to everyone. Inexpensive commodity hardware sits idle much of the day, and almost completely idle at night. Of the several problems with using desktop machines, one of the more difficult is that the real users occasionally want to use the machine themselves.

Removing one machine represents only a minor drop in the total computational power, but if not properly handled the entire computation could stall waiting for data from that machine. However, migration allows the objects on that machine to be moved elsewhere, which allows the computation to proceed.

3.5.2 Shrinking and Expanding the Set of processors

For timeshared parallel machines, migration allows an adaptive job scheduler to be built that allows jobs to change the number of processors they use at runtime. This "shrink/expand" capability has been demonstrated to improve system utilization and the average response time of the jobs[10].

With adaptive job scheduling, when a job enters the system it specifies the minimum number of processors (typically from memory usage considerations) and maximum number of processors (typically from scalability considerations) that the job can use. A fair adaptive job scheduler will first allocate all jobs their minimum number of processors, then allocate any remaining processors among the jobs. If a job cannot be allocated its minimum number of processors, or if another job has higher priority, the leftover job will be put in a wait queue. When a job finishes running, the scheduler reanalyzes the distribution of processors and some jobs may shrink or expand.

3.6 Communication Optimization

With processor speeds rising faster than communication latency is dropping, communication optimizations are becoming crucial to high performance. Though this has been an active area of research over the past decade, the main emphasis of this research has been on processor to processor communication optimizations. These communication calls tend to be synchronous and the aim is to reduce the total time of the communication operation—for example, there has been much work on improving broadcasts, reductions, and all to all personalized communication.

In a virtualized program, communication operations happen at the object level. This means communication operations should be asynchronous, with the aim of minimizing the time the processor spends on the operation to allow another object to run. Since many fast parallel machines have communication co-processors, the main processor should have as little work to do in communication as possible.

In a fully virtualized scenario thousands of objects may reside on one processor. Grouping messages these objects send to the same processor would save on message startup costs, and could improve system performance. However, grouping messages also affects the pipelining of computation and communication, so there is a trade-off.

Grouping messages could also improve the performance of collective communication operations like all to all personalized sends. The direct implementation of this operation, where every object sends to every other object, leads to heavy network contention which can make this operation costly and not scalable. With message combining, all the objects on a processor send a combined message to a subset of their destination processors, from where they are routed to their final destinations. For example, in a 2D mesh virtual topology (with $\sqrt{P} \times \sqrt{P}$ processors) each processor sends $\sqrt{P}$ combined messages (each consisting of $\sqrt{P}$ individual messages) to the processors in its column in the 2D mesh. On receiving messages from all the processors in its column, each processor extracts and combines messages destined to each processor in its row and sends it as one message. Other topologies like 3D mesh and hypercube can also be imposed, although the best strategy depends on the application.

In general, virtualization gives the runtime system an opportunity to collect performance data on the objects, and then to choose the best communication method at runtime. Virtualized communication optimizations also have to handle streaming and aperiodic messages. These messages are not cyclic and may not appear at regular intervals of time.

Charm++ already provides efficient implementations of per-object broadcasts and reductions. We are currently developing more advanced communication optimizations, and more importantly the methods for choosing the appropriate optimization for a particular application.

4 Components: Charisma

Developing large scale parallel applications requires integration of independent software modules, each perhaps employing a different set of parallel libraries. Such a coupling is often associated with high communication costs that might hamper the scalability of the program. In-process[1] components [14] eliminate much of the

[1]Library objects running within the same process as its client.

inefficiency of languages like CORBA; do not require the a serialized data exchange; and can be made only slightly more expensive than a procedure call.

We have developed a general purpose component architecture, *Charisma*, for parallel applications with in-process components [2]. *Charisma* is built on Charm++, which provides a common language runtime and allows an application to have multiple in-process components. The *Charisma* interface model, based on data-driven control transfer, allows components to access other components' functionality in a uniform manner.

In this interface model, each component describes the output data it publishes, and the input data it accepts on a set of named and typed "ports." A component's input arrives via a set of input ports. The data a component produces is published on its output ports. The connection between input ports and output ports is kept outside of either component's code, in a separate connection specification.

In AMPI, input and output ports are simply special, additional MPI communicators, and the usual range of MPI communication can be used to receive data from input ports and send data to output ports. In Charm++, input ports are ordinary methods with a special tag, and output ports are attached to a special kind of object.

With ports, individual component codes can be considered in isolation, developed more easily, and re-used in disparate contexts. Since the connection specification for an application is all in one place, even dramatic connection rearrangements can be made from one centralized location, rather than having to edit dozens of individual codes.

5 Domain-Specific Frameworks

There seem to be three sensible alternatives for making parallel computing more useful—more automatic analysis, higher-level general-purpose languages, and task-specific frameworks. Many workers have attempted to automatically extract parallelism from serial code, and similarly higher-level parallel languages abound. Task-specific frameworks, however, can combine both dramatic serial code reuse with a natural means to express parallelism.

The distinction between a parallel library and a parallel framework is somewhat arbitrary, but a framework generally determines the overall structure of the program.

For example, the Charm++ Finite Element Method (FEM) Framework [1] provides a robust environment for parallelizing FEM computations. At startup, the user passes the FEM mesh into the framework, where it it partitioned into pieces. After partitioning, user code runs on the partitions, occasionally calling the framework to communicate between partitions. Because the basic form of an FEM program is known, the communication facilities in the framework very closely match the application developer's view of the computation. This results in a much shallower learning curve than a general-purpose interface like MPI or Charm++.

In addition to ease of development, a framework also provides a natural layer in which to apply optimizations. For example, the FEM framework is a natural place to do message combining for communication optimizations, or to experiment with a new method for partitioning FEM meshes.

6 Conclusions

A critical, and far from obvious choice in any system is the proper abstraction boundary. In parallel computing, this choice determines the division of labor between the application developer and the runtime system developer. We believe that today's parallel runtime systems like MPI are too low level, which forces application developers to reinvent or do without useful capabilities. Among the capabilities missing from MPI, but present in advanced runtime systems like Charm++ and AMPI, are message-driven execution, automatic load balancing, checkpointing, and high-level communication optimizations. With carefully chosen abstractions, a runtime system can enable applications to be developed easily, be flexible, and above all be efficient.

References

[1] 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.

[2] M. Bhandarkar and L. V. Kale. An Interface Model for Parallel Components. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC), Cumberland Falls, KY*, August 2001.

[3] 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.

[4] 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.

[5] L. Kalé and S. Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[6] L. 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] L. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[8] L. V. Kale, M. Bhandarkar, and R. Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[9] 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.

[10] L. V. Kalé, S. Kumar, and J. DeSouza. An adaptive job scheduler for timeshared parallel machines. Technical Report 00-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Sep 2000.

[11] 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.

[12] M. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L. Kale, R. D. Skeel, and K. Schulten. NAMD—a parallel, object-oriented molecular dynamics program. *Intl. J. Supercomput. Applics. High Performance Computing*, 10(4):251–268, Winter 1996.

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

[14] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[15] N. 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.