Using Message-Driven Objects to Mask Latency in Grid Computing Applications

Gregory A. Koenig and Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
{koenig,kale}@cs.uiuc.edu

Abstract

One of the attractive features of Grid computing is that resources in geographically distant places can be mobilized to meet computational needs as they arise. A particularly challenging issue is that of executing a single application across multiple machines that are separated by large distances. While certain classes of applications such as pipeline style or master-slave style applications may run well in Grid computing environments with little or no modification, tightly-coupled applications require significant work to achieve good performance.

In this paper, we demonstrate that message-driven objects, implemented in the Charm++ and Adaptive MPI systems, can be used to mask the effects of latency in Grid computing environments without requiring modification of application software. We examine a simple five-point stencil decomposition application as well as a more complex molecular dynamics application running in an environment in which arbitrary artificial latencies can be induced between pairs of nodes. Performance of the applications running under artificial latencies are compared to the performance of the applications running across TeraGrid nodes located at the National Center for Supercomputing Applications and Argonne National Laboratory.

1. Introduction

Due to the growth of distributed Grid computing technologies and environments [8, 9] over the past several years, consumers of high performance computing cycles are increasingly considering the feasibility of deploying applications that span multiple geographically distributed sites. Software such as Globus [7] allows the creation of so-called "virtual organizations" in which computational resources owned by multiple physical organizations are united to form a single cohesive resource for the duration of a single computational job.

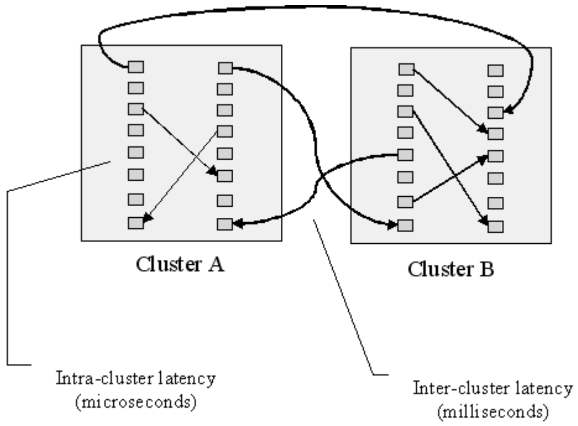


Figure 1. An example of an application co-allocated across two clusters

One of the fundamental challenges to deploying Grid applications across geographically distributed computational resources is overcoming the effects of the latency between sites. While the interconnects used in contemporary high-performance clusters and supercomputers can deliver data to applications with latencies on the order of a few microseconds, latencies across the wide-area are usually measured in milliseconds. Figure 1 illustrates this idea. Certain classes of applications lend themselves well to running in such an environment. Pipeline style applications such as those that do simulation on one cluster and visualization on another typically have very coarse granularity and accordingly can tolerate latency well. Master-slave style applications are also good candidates for Grid environments because they typically have small communication requirements and because communication delays are often not on the critical path.

In contrast, some classes of applications present serious challenges to deployment in Grid computing environments.

For example, tightly-coupled applications where processors communicate with each other during every iteration present a significant challenge. Masking the effects of wide-area latency is critical for achieving good performance with these types of Grid applications that involve non-trivial amounts of communication. To date, most work involving the deployment of tightly-coupled applications on computational Grids has focused on algorithm-level modifications necessary for latency tolerance.

In this paper, we demonstrate that message-driven objects, implemented in the Charm++ and Adaptive MPI systems, can be used to mask the effects of latency in Grid computing environments when tightly-coupled applications are co-allocated across geographically distributed resources. Because the technique is encapsulated within the runtime layer, it can be applied to a wide variety of problem decomposition strategies, such as regular and irregular mesh decomposition or spatial decomposition, without requiring modification of application software. Furthermore, through the use of Adaptive MPI, any MPI application can take advantage of our techniques. The remaining sections of this paper describe the enabling technologies that provide a basis for our work, the relationship between our work and other work in this area, and the experimental results of applying our technique to two example applications. Performance of the example applications, a simple five-point stencil decomposition application and a more complex molecular dynamics application, is examined in an environment in which arbitrary artificial latencies can be induced between pairs of nodes. Performance for the applications running under artificial latencies is compared to the performance of the applications running across TeraGrid nodes located at the National Center for Supercomputing Applications and Argonne National Laboratory.

2. Enabling Technologies

In this section, we describe the enabling technologies upon which our work is based. These technologies include the Charm++ runtime system and the Virtual Machine Interface message layer.

2.1. Charm++ and AMPI

Charm++ [16] is a message-driven parallel programming language designed with the goal of enhancing programmer productivity by providing a high-level abstraction of a parallel computation while at the same time providing good performance on platforms ranging from traditional supercomputers to more recent commodity cluster environments. Charm++ is based on the C++ programming language and is backed by an adaptive runtime system that provides features such as processor virtualization, load balancing, and optimized communication libraries, especially for collective operations such as broadcasts and reductions.

Programs written in Charm++ consist of parallel objects called *chares* that communicate with each other through asynchronous message passing. When a chare receives a message, the message triggers a corresponding method within the chare object to handle the message asynchronously. Further, chares may be organized into one or more indexed collections called *chare arrays*. Messages may be sent to individual chares within a chare array or to the entire chare array simultaneously.

The chares in a Charm++ program are assigned to processors by an adaptive runtime system. The runtime system may also change this assignment dynamically by migrating chares among processors. A suite of measurement-based load balancers is provided to take advantage of this capability. In addition, the migration capability is leveraged to support other capabilities such as automatic checkpointing, fault tolerance, and the ability to shrink and expand the set of processors used by a parallel job.

Adaptive MPI (AMPI) [15] provides the same capabilities as Charm++ in a more familiar MPI programming model. AMPI implements the MPI standard by encapsulating each MPI process within a user-level migratable thread. By embedding each thread within a Charm++ object, AMPI programs can automatically take advantage of the features of the Charm++ runtime system with little or no changes to the underlying MPI program.

One of the central enabling ideas in Charm++ and AMPI is that of having the programmer divide the computation into a large number of objects, sometimes thought of as virtual processors. Because there are typically multiple objects per physical processor, a scheduler that sequences object execution based on availability of messages is naturally needed. This model of *message driven execution* leads directly to some performance benefits such as automatic adaptive overlap of communication and computation. The message-driven execution model is similar to the model used in systems such as Active Messages [32], Fast Messages [27], and Nexus [10, 11]; Charm and its predecessor Chare Kernel [18] is contemporaneous to or precedes the development of these models.

The idea of using multiple virtual processors per physical processor appears in the literature early. For example, Fox's book [12] describes the use of virtualization to randomize the placement of sub-blocks for load balancing. Virtualization techniques were also used in the CM-2 [2] and in DRMS [23, 24]. Charm++'s contribution is in providing an adaptive runtime system, including measurement based load-balancing, that exploits the degree of freedom provided by virtualization.

2.2. Virtual Machine Interface

The proliferation of high-performance clusters built from commodity off the shelf components has resulted in the widespread deployment of several high-bandwidth low-latency networks such as Myrinet [5] and InfiniBand [30]. The Virtual Machine Interface (VMI) message layer [29, 28] was designed to be a low-cost abstraction layer providing compatibility across multiple interconnects. Using software modules that are dynamically loaded at runtime, VMI allows applications to be switched from one interconnect to another without requiring the application to be recompiled or re-linked. Further, by organizing these dynamically-loaded software modules into send and receive *chains* of modules, novel capabilities can be developed at the messaging layer level. For example, by loading multiple modules simultaneously, data may be striped across multiple interconnects. Also, an application can run in a Grid computing environment using high-performance networks to communicate with local neighbors within a computation and wide-area networks to communicate with neighbors located on remote nodes. Finally, because modules can intercept and manipulate message data as it is passed from module to module, capabilities such as encrypting or compressing the data are possible.

VMI is not typically intended to be a software layer exposed to application developers, but rather as a layer upon which higher level message layers or runtime systems can be built. To this end, an efficient implementation of Charm++ that uses VMI as its underlying message passing layer has been developed [20]. This version of Charm++ is used for all experiments described in this paper.

3. Related Work

The work we describe in this paper shares characteristics with several other projects while at the same time offering its own unique contributions. In this section, we describe related work and draw comparisons and contrasts to our work.

Viewing a distributed computation as a set of interacting objects and, accordingly, using Object Oriented Programming techniques to manage complexity is an attractive approach to Grid computing. Several other projects such as Legion [13] and Globe [31] share this characteristic with our work. In contrast to our work, however, both of these projects tend to be focused more on the entire range of problems surrounding Grid computing, including resource management, file and data access, information brokering, and security. Indeed, the designers of Legion call such an all-encompassing system a *metasystem* while the designers of Globe have the goal of building a middleware layer capable of scaling to billions of users around the world. Examples of applications running in these kinds of environments appear to be focused on those types that can implicitly tolerate latency, such as parameter sweep applications [26], or applications such as molecular dynamics applications running entirely within the context of a single machine on the Grid [25]. Our goal differs from this type of work in that we are specifically focused on the topic of developing techniques for masking latency for tightly-coupled applications running across multiple machines connected via a Grid.[1]

Several projects extend the MPI parallel computing standard to work in a Grid environment with the goal of allowing jobs that can span multiple clusters. Examples of such projects include MPICH-G2 [19] and MPICH/MADIII [4]. These projects, like ours, view the communication infrastructure of a distributed Grid job as a hierarchy of interconnects. Such a view consists of, for example, high-performance interconnects such as Myrinet or InfiniBand (used for intra-cluster communication between processes co-allocated on the same cluster) at one level of the hierarchy and lower-performance wide-area interconnects such as TCP/IP (used for inter-cluster communication between processes on different clusters) at another level of the hierarchy. The goal is to allow any pair of processors to communicate via the most efficient channel possible within the hierarchy. MPICH-G2 achieves this goal by relying on whatever underlying native MPI implementation that exists on a cluster to provide efficient intra-cluster communication and TCP/IP to provide inter-cluster communication. MPICH/MADIII takes an approach that is very similar to our implementation of Charm++ on VMI. MPICH/MADIII is implemented on top of a communication library, Madeleine III, that allows multiple underlying networks to be used in a way similar to VMI. Further, MPICH/MADIII uses an efficient user-level thread library, Marcel, that provides task decomposition capabilities similar to what is available with AMPI threads or Charm++ chares. In contrast to MPICH/MADIII, however, our work seeks to increase the number of opportunities for the adaptive runtime system to overlap useful computation with communication by using very large degrees of virtualization, with perhaps thousands of objects per physical processor in the computation. MPICH/MADIII seems to typically use a much smaller number of threads per processor. Further, the Charm++ adaptive runtime system includes the ability to dynamically load-balance objects within a distributed computation while MPICH/MADIII does not seem to offer this functionality. We believe that this capability is critical to achieving good performance on fine-grained Grid computations that span multiple clusters.

Various algorithm-level approaches to tolerating latency in Grid computing environments exist. For example, the authors of [6] describe a technique for improving the per-

[1] Although it should be noted that we are also involved in a project called Faucets [17] that has the goal of efficient resource management and scheduling of computational resources within a Grid environment.

formance of Partial Differential Equation solvers running in a Grid environment by increasing the number of ghost cell layers used per processor. Increasing the number of ghost zones allows each processor to buffer more data and reduces the number of messages sent between processors. Further improvements to the PDE algorithm allow the elimination of diagonal communications. Together, these algorithm-level optimizations allow the performance of the application described in the paper to be improved by as much as 170%. The primary contrast between approaches such as the one described in the paper and our work is that we are attempting to provide techniques that operate at the runtime layer rather than at the application layer. While algorithm-level approaches have the advantage that they can generally achieve very good levels of optimization, runtime-level approaches have the advantage that they are automatically available to developers and do not require modification of application software. Further, the technique of increasing the number of ghost zone layers is a pattern-specific technique that is not applicable to all problems such as the LeanMD molecular dynamics code described later in this paper.

Finally, Cactus-G [3] is a Grid-enabled computational framework that is based on the Cactus problem solving environment and the MPICH-G2 message passing library. Originally designed for use in numerical relativity applications such as modeling black holes, neutron stars, and gravitational waves, Cactus has grown into a framework well-suited to solving general mesh decomposition problems. A novel feature of Cactus is that it consists of a central core called the *flesh* which connects to application modules called *thorns* through an extensible interface. The thorns in a computation encapsulate the actual scientific code governing the application as well as capabilities such as parallel I/O, data distribution, and checkpointing. The experiment described in the paper leverages this rich platform to synthesize a heterogeneous environment consisting of four machines distributed between the San Diego Supercomputing Center (SDSC) and the National Center for Supercomputing Applications (NCSA) to run a tightly-coupled mesh decomposition problem. The authors are successful in this endeavor due to the ability to leverage thorns that optimize the computation in three ways. First, because the resources physically allocated to the computation consisted of one machine at SDSC and three machines at NCSA, the authors positioned the gridpoints in the mesh to reflect this uneven distribution. Second, the authors increased the size of the ghost zone layers on each processor similar to the method used in [6] above. Third, the authors used a thorn to compress message data that were sent over the wide-area connection between SDSC and NCSA. In many ways, Cactus-G can be thought of as an elaborate runtime system that offers features similar to those found in the Charm++ runtime. Our work differs from the work in Cactus, however, in that our approach is to focus on dividing a computation into a large number of Charm++ objects or AMPI threads and then to dynamically map and re-map these entities onto physical processors as the computation progresses. In some sense, our approach is at a lower level than the approach taken by Cactus-G.

The common thread that differentiates our work from others is our use of processor virtualization, in the form of Charm++ chares or AMPI threads, as a means of tolerating latency in Grid computing environments without requiring modification of application software. To this end, we examine this concept in greater detail in Section 4.

4. Latency Tolerance in Grid Applications

The fundamental idea behind the Charm++ runtime system is that a programmer divides a program into a large number of message-driven objects, implemented in the form of either Charm++ chares or AMPI threads. In some sense, these entities virtualize the notion of work, so it is sometimes also convenient to think of them as virtual processors. The number of virtual processors (objects) is independent of, and in practice much larger than, the number of physical processors. The runtime system maps virtual processors onto physical processors and may dynamically adjust this mapping as the application executes to balance load or optimize communication costs. Rather than thinking in terms of physical processors, the programmer thinks in terms of the virtual processor abstraction and writes code to coordinate interactions among these virtual entities. These interactions are realized as asynchronous messages that are passed between physical processors in the computation. As messages arrive at a physical processor, they are enqueued in a message queue in either FIFO or priority order. When a physical processor becomes idle, its message scheduler dequeues the next waiting message and delivers it, triggering the execution of code that is encapsulated within an object to handle the message. This code runs to completion, producing other messages for objects on this or another physical processor.

Because messages are asynchronous, the runtime system may schedule the execution of a new object immediately after execution within an existing objects completes, resulting in one or more messages sent by the object. Rather than waiting for these messages to be delivered, the newly-scheduled object begins work immediately, thus overlapping its computation with the communication of the previous object. This ability to overlap useful computation with communication is important within the context of a single cluster, but is especially critical for applications that are colocated across multiple clusters separated by a high-latency wide-area network. Figure 2 illustrates this concept by de-

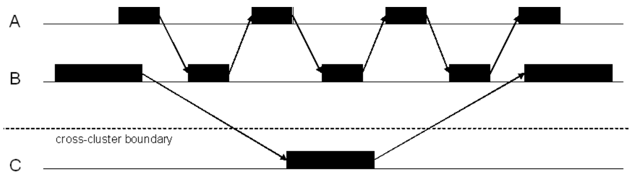


Figure 2. A hypothetical timeline illustrating the use of message-driven objects to tolerate wide-area latency

picting a hypothetical timeline for three processors running on two clusters that are connected by a high-latency wide-area network. Processors A and B are located within one cluster and processor C is located within the second cluster. At the left side of the timeline, processor B sends a message to processor C; this message must cross the high-latency inter-cluster network. Rather than waiting idly for this message to be delivered, B is free to respond to an incoming message from processor A, and in fact performs several short computations and message exchanges with A. Finally, processor C responds to processor B with the result of the computation previously triggered by B. The important idea is that B is able to do useful work during the gap between dispatching a message to C and receiving a response.

For illustrative purposes, we consider the effectiveness of this technique for masking wide-area latency experienced by two example applications that are co-allocated across two clusters on a computational Grid. In Section 5, we give specific details regarding experimental results for these applications.

The first application we consider is a simple five-point stencil finite difference method. In this class of numerical method, a multidimensional mesh is repeatedly updated by replacing the value at each point with some function of the values at a small, fixed number of neighboring points. In this case, the neighboring points taken into consideration are the ones directly above and below as well as to the left and right of a given cell. This produces four discrete communication events per cell for each time-step. For the work in this paper, we consider a mesh of size 2048x2048 cells. The problem is decomposed using virtualization by dividing the cells within the mesh evenly among a specified number of objects. For example, for a 2048x2048 mesh divided among 64 objects, 8 objects are mapped along each axis of the mesh. Accordingly, each object has a 256x256 square section of the mesh to operate upon. During each time step, each object communicates values for a 256x1 vector of cells to its appropriate neighbor. Because the problem is split across two clusters separated by a wide-area connection, every time step involves some processors communicating with neighbors situated across the wide area. More importantly, however, each cluster contains a large number of processors that communicate with neighbors solely within the local cluster. The expectation is that the message-driven execution model will allow the high-latency communication operations to be masked by other communication that is carried out with neighbors situated on the local cluster. The five-point stencil is an attractive problem to consider because it allows us to readily choose a varying degree of virtualization by increasing or decreasing the number of objects used to decompose the mesh.

The second application we consider is a more complex, classical molecular dynamics code called LeanMD [22]. Within a LeanMD simulation, the atoms of a biomolecular system, including proteins, cell membranes, SNA, and waters, are partitioned into a group of cells. Electrostatic (and van der Waal's) interactions between every pair of neighboring cells are computed by a separate cell-pair object. These interactions constitute the bulk of processor time used by the application, although there are other force computations involving bonds between atoms. In each time-step, each cell "integrates" all forces on its atoms, and changes their positions based on new acceleration and velocities calculated. It then multicasts its atom's coordinates to the 26 cell-pairs that depend on it. Each cell pair calculates forces on the two sets of atoms it receives, and sends them back to the two cells. In the context of this paper, it is important to note that the computations in each cell pair depends on messages from at most two other objects, possibly on two different processors. Thus, in the benchmark used in this paper (Section 5), there are 216 cells and 3,024 cell pairs. On each processor, there may be several tens of cell-pair objects. In a multi-cluster context, some of subset of these objects ("subset A") require messages from cells within their own cluster, while a different subset ("subset B") may require one or both messages from outside the cluster. As a result, a processor is able to execute objects in subset A while waiting for high-latency messages for objects in subset B from another cluster. This renders the application latency tolerant to some extent. Although the degree of virtualization can be varied much more readily in applications such as the stencil decomposition described above, LeanMD is an attractive application to consider because it is more representative of realistic scientific codes.

The key contribution of this paper is the demonstration that the use of parallel message-driven objects, as implemented in systems such as Charm++ or AMPI, can mask the high latencies found in Grid computing environments when tightly-coupled applications are co-allocated across distributed resources.

5. Experimental Results

In this section, we describe a set of experiments based on the five-point stencil decomposition application and LeanMD application described in Section 4. Results of these experiments demonstrate that virtualization of work can be used as a mechanism for masking latency in Grid computing environments.

5.1. Experimental Environment

All experiments described in this paper are carried out in two environments, both consisting of a pair of clusters. All cluster nodes in both environments are dual-processor Itanium 2 machines running at 1.5 GHz and containing 4 GB of main memory each. For each experiment conducted, the number of physical processors used for the experiment is varied in increasing powers of 2 (i.e., 2, 4, 8, 16, 32, and 64 processors) and these processors are evenly distributed between the two clusters (i.e., 1+1, 2+2, 4+4, 8+8, 16+16, and 32+32 processors). The result is that half of the processors allocated to the application are physically located on one cluster and the other half on the second cluster, with messages sent between co-allocated processors going over a high-latency interconnect.

The first environment used for experiments is a "simulated Grid environment" physically consisting of nodes that exist within a single real cluster.[2] In this simulated Grid environment, arbitrary latencies can be inserted between any pair of nodes, allowing us to sweep cross-cluster latencies across a range to study the impact of varying wide-area latencies on the underlying application. Recall from Section 2.2 that the Virtual Machine Interface messaging layer is used for all communication operations described in this paper, and that a novel feature of VMI is the ability to organize the device drivers used for these communication operations into send and receive chains of drivers. As message data travels along a chain, each driver on the chain examines the message to determine whether that driver should deliver the message or whether it should simply send the message to the next device in the chain for eventual delivery by some lower-level device. Furthermore, a device driver may manipulate the message in arbitrary ways. We leverage this capability to inject pre-defined latencies between arbitrary pairs of nodes by constructing send and receive chains that consist of two network drivers with a "delay device driver" in between. By affiliating a subset of the cluster's nodes with the first driver in the chain, message data are immediately sent between the nodes within that subset without passing through the delay device. For nodes not in this affiliation (i.e., those that exist on the "remote cluster"), messages are intercepted by the delay device which delays the message by a pre-defined amount of time before passing it to the network device driver used to communicate over the "wide area."

The second environment used for experiments is a true Grid computing environment composed of TeraGrid [1] resources located at the National Center for Supercomputing Applications and at Argonne National Laboratory.[3] ICMP ping latencies between these clusters are reported as approximately 1.725 ms one-way latency, and simple Charm++ ping-pong latencies are approximately 1.920 ms.

5.2. Results for Five-Point Stencil

Figure 3 shows results for the five-point stencil experiment running on a 2048x2048 mesh. Separate sub-graphs show the performance of the application for numbers of physical processors ranging from 2 to 64 processors. Because the problem is of a fixed size, always 2048x2048 cells, as the number of physical processors increases, the granularity of the portion of the problem residing on each processor decreases.

The first thing to observe is that for instances of the problem with relatively large grain size (e.g., for 2 and 4 processors in Figures 3(a) and 3(b)), the execution time for several different degrees of virtualization remains almost constant. That is, the near-horizontal lines on the graphs are significant because they indicate that execution time stays near constant as latency increases. As the number of processors increases and the granularity of the problem begins to decrease, latencies above a certain point are no longer able to be tolerated as well. This is shown on the graphs as plots with near-horizontal sections followed by a section of increasing execution time (e.g., Figures 3(c), 3(d), and 3(e)). Finally, for the graph corresponding to 64 processors in Figure 3(f), the near-horizontal sections of the plots are very short. The important thing to note from this figure is that the near-horizontal sections for plots corresponding to higher degrees of virtualization (i.e., 256 or 1024 objects) are longer than those of the plot corresponding to a lower degree of virtualization. Moreover, as the system begins to become less able to tolerate the latency between clusters, the slope of the plots corresponding to higher degrees of virtualization is less steep than the slope of the plot corresponding to the lower degree. That is, the experiments that used higher degrees of virtualization were better able to mask the effects of wide-area latency, even for latencies that are very large in comparison to the step time. These results match predictions made in other work [14].

In the graphs in Figures 3(b), 3(c), and 3(d), the results for the smallest degree of virtualization show markedly

[2] This is the NCSA TeraGrid cluster, tg-login.ncsa.uiuc.edu, also known as Mercury.

[3] These are the machines tg-login.ncsa.teragrid.org and tg-login.uc.teragrid.org, respectively.

Processors	Objects	Time (ms/step) Artificial Latency	Time (ms/step) Real Latency
2	4	85.774	96.597
2	16	75.050	79.488
2	64	80.436	77.170
4	4	85.095	90.815
4	16	35.018	35.546
4	64	36.667	37.345
8	16	25.468	26.237
8	64	17.596	18.444
8	256	19.853	20.853
16	16	17.114	17.752
16	64	10.959	11.588
16	256	10.017	10.913
32	64	6.756	7.405
32	256	6.022	6.622
32	1024	8.090	8.090
64	64	6.708	7.364
64	256	3.963	4.459
64	1024	4.928	4.906

Table 1. Comparison of five-point stencil execution times running under artificial latency and across multiple clusters

worse performance than for higher degrees of virtualization. We believe that the performance improvements with higher degrees of virtualization are due to improved cache performance because of smaller grainsize. This is an area of ongoing investigation.

As a means of validating the results collected in the simulated Grid environment, Table 1 shows a comparison between the results collected under artificial latencies and in the real Grid environment between NCSA and Argonne. For many data points, the real multi-cluster results match the results collected under artificial latency very well. More importantly, the trends in the data collected under artificial latency correspond to the trends in the data collected in the real Grid environment.

5.3. Results for LeanMD

As explained earlier, LeanMD is a tightly coupled parallel program with a significant complexity in its communication structure. We conducted two sets of experiments to demonstrate the extent of latency tolerance achieved with our approach: one with artifically added latencies and one with remote clusters.

Figure 4 shows the performance of LeanMD as a function of message latency on different number of processors. The latency is varied from 1 to 256 milliseconds. Each computation step is about 8 second on a single processor. The natural parallelism of the application leads to reasonable scaling up to 32 processors, as seen by examining the leftmost points on each curve. On 64 processors, the speedup starts to stagnate. (The runs were conducted without any load balancing. With load balancing, the speedups are likely to be good at 64 processors, but they will eventually stagnate due to natural limit of application parallelism [22]. The main point of these experiments is not the speedups per se, but the impact of latency on speedup.) On 2 processors, latency makes almost no impact, in part because even with 256 ms latency at the right end of the curve, the latency is a fraction of the step time. But even here, it is worth noting that with a round trip latency of 512 ms (0.5 seconds), many algorithms would have increased their per-step time from 4 to 4.5 seconds at least. However, the data for 32 processors is even more impressive: with a per-step time as short as 300 ms, the graph shows no impact of latency as high as 32 ms. This latency tolerance is achieved by virtue of the large number of objects created by the application: over 3,000 cell-pair objects divided among 32 processors leads to over 90 objects per processor. Since many of these objects depend only on local-cluster messages, the wait for remote-cluster messages is automatically overlapped with useful computation by Charm++'s message-driven scheduler.

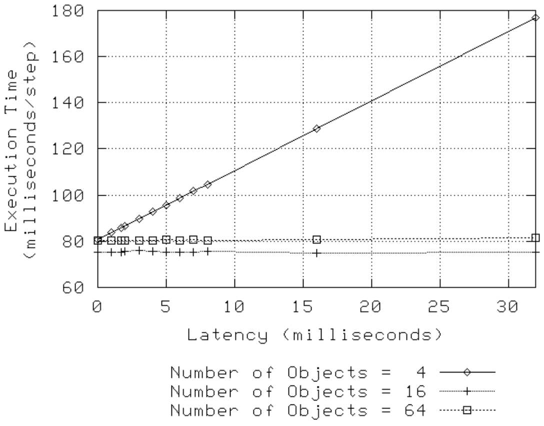
Processors	Time (ms/step) Artificial Latency	Time (ms/step) Real Latency
2	3.924	3.924
4	2.021	2.022
8	1.015	1.018
16	0.559	0.550
32	0.302	0.299
64	0.239	0.260

Table 2. Comparison of LeanMD execution times running under artificial latency and across multiple clusters

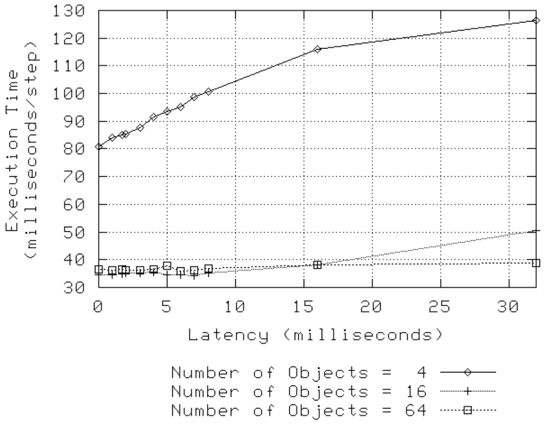
To validate these results, we ran the same benchmark on the two remote TeraGrid clusters mentioned above. The results are shown in Table 2. For up to 32 processors, the real multi-cluster results match extremely well with what can be predicted based on our artificial-latency experiments. For 64 processors, the correspondence is still good, but not as good as that for fewer processors. We speculate that this may be due to the fact that latencies will be higher when a large amount of data is being communicated between two clusters over a shorter period of time, leading to increased contention in the network.

6. Conclusion

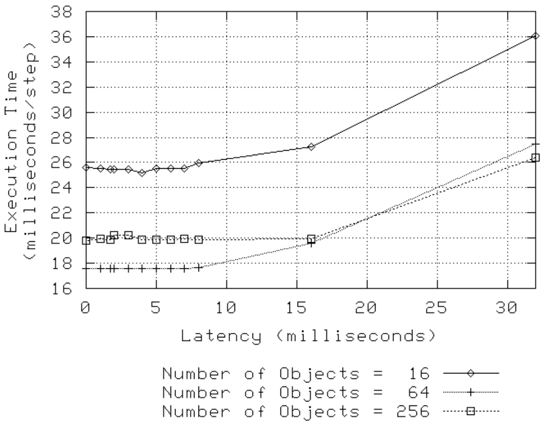
In this paper, we have demonstrated that message-driven objects, implemented in the Charm++ and Adaptive MPI systems, can be used to mask the high latencies found in Grid computing environments when tightly-coupled appli-



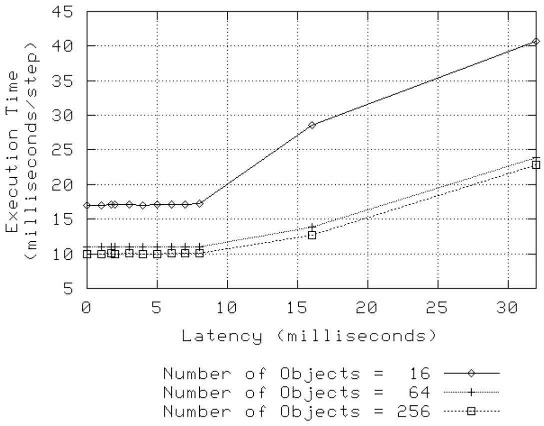
(a) Processors = 2



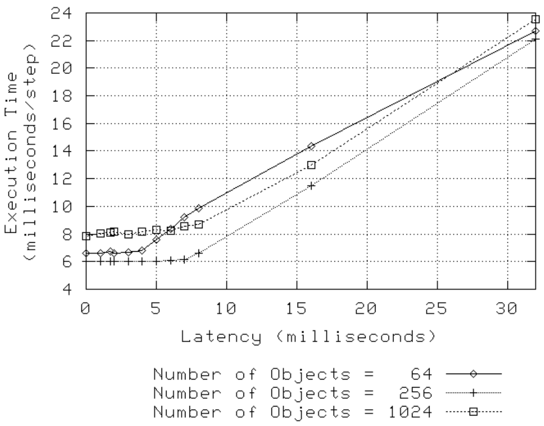
(b) Processors = 4



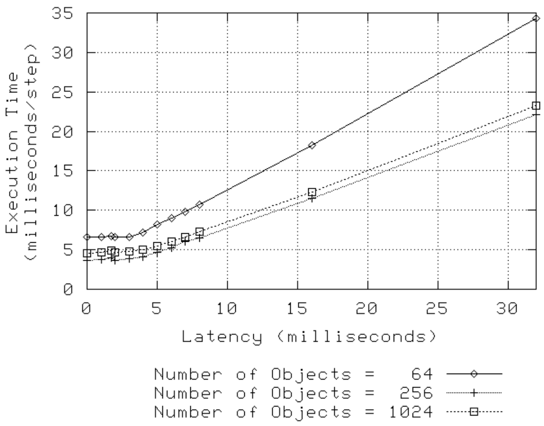
(c) Processors = 8



(d) Processors = 16



(e) Processors = 32



(f) Processors = 64

Figure 3. Performance of five-point stencil with artificial latencies 0-32 milliseconds

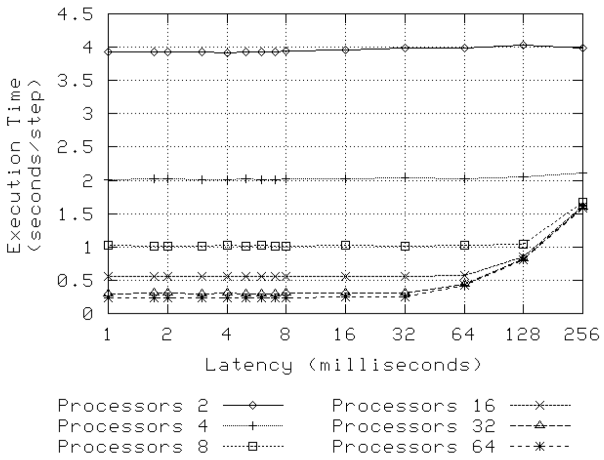


Figure 4. Performance of LeanMD running Human Carbonic Anhydrase simulation with artificial latencies 1-256 milliseconds

cations are co-allocated across geographically distributed resources. Further, because the technique is encapsulated within the runtime system, it can be applied to a wide variety of problem decomposition strategies, such as regular and irregular mesh decomposition or spatial decomposition, without requiring modification of application software. This is in contrast to algorithm-specific techniques for latency masking that are often applicable to only a particular decomposition strategy.

Based on the preliminary technique described in this paper, we intend to perform further experimentation and investigation in several areas. First, we intend to further verify the experimental results obtained under artificial latencies for the five-point stencil and LeanMD applications by comparing the results against performance results gathered by running the applications on TeraGrid nodes located at the National Center for Supercomputing Applications (NCSA) and at the San Diego Supercomputing Center (SDSC). One-way latency between these sites is approximately 29.37 milliseconds. Accordingly, we expect that example codes with larger per-step execution times should be able to run successfully in this environment. We further expect that example codes such as the five-point stencil running over a 2048x2048 mesh will experience severe performance penalties as suggested by our experiments using artificially induced latencies.

Second, we have begun work on a Charm++ load balancer specifically designed for Grid computing environments. The preliminary version of this load balancer uses the strategy of simply distributing the chares that communicate across high-latency wide-area connections evenly among the processors within a cluster. In this scheme, no chares are migrated to remote clusters; rather they are simply migrated among the processors within the cluster in which they were originally placed.

Third, the Charm++ and AMPI systems support the notion of prioritized message delivery. Under this scheme, messages are tagged with a priority and are delivered to the application in order of decreasing priority. This feature is attractive for our work because one can envision a scheme in which messages that cross cluster boundaries are tagged with a higher priority than local messages. This tagging would allow these messages to be processed first, further reducing the impact of wide-area latency on the application.

Fourth, we believe that leveraging Adaptive MPI is an important aspect of our future work due to the large number of MPI applications that exist, representing a wide variety of problem types. AMPI programs can take full advantage of Charm++ runtime features including load balancing, checkpointing and fault tolerance, and communication optimizations. Due to this ability to readily leverage existing Charm++ features, we expect that using AMPI for our future investigations in Grid environments to be straightforward.

Finally, and perhaps most importantly, our future work in using parallel message-driven objects as a mechanism for providing latency tolerance to applications running across multiple clusters in a Grid environment will be underscored by two primary scenarios in which we envision the work as being directly applicable. One scenario is running extremely large computations that exceed the capacity of a single cluster. For example, applications such as finite element models are often bounded by the total amount of memory available, with scientists seeking to run on as many nodes as possible to gain access to large amounts of memory. By synthesizing the resources in two or more clusters, this goal can be more readily achieved. A second scenario is scheduling computational resources on-demand, in which a job is submitted along with a deadline by which the job must be completed [21]. To fulfill such a requirement in cases where no single computational resource has sufficient capacity to fulfill the request by the given deadline, a job request might be satisfied by allocating some nodes from one cluster and the balance of nodes needed by the job from a second cluster. This might be useful in circumstances in which the request for nodes cannot be satisfied by a single cluster by the deadline specified by the job. Such work fits well with the goals of our work with the Faucets [17] Grid computing environment.

References

[1] TeraGrid project homepage. http://www.teragrid.org/.

[2] Connection Machine model CM-2 technical summary. Technical report, Thinking Machines Corporation, 1990.

[3] G. Allen, T. Dramlitsch, I. Foster, N. T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. *Proceedings of SC2001*, November 2001.

[4] O. Aumage and G. Mercier. MPICH/MADIII: a cluster of clusters enabled MPI implementation. *Proceedings of 3rd International Symposium on Cluster Computing and the Grid*, May 2003.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. K. Su. Myrinet — A gigabit-per-second local-area-network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving PDE problems. *Proceedings of SC2001*, November 2001.

[7] I. Foster and C. Kesselman. The Globus toolkit. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 259–278. Morgan-Kaufmann, San Francisco, CA, 1999.

[8] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan-Kaufmann, July 1999.

[9] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a Future Computing Infrastructure*. Morgan-Kaufmann, January 2004.

[10] I. Foster, C. Kesselman, and S. Tuecke. Nexus: Runtime support for task-parallel programming languages. Technical Memo ANL/MCS-TM-205, Argonne National Laboratory, 1995.

[11] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[12] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume Volume I: General Techniques and Regular Problems. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[13] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and J. Paul F. Reynolds. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, University of Virginia Computer Science Department, June 1994.

[14] A. Gursoy. *Simplified Expression of Message Driven Programs and Quantification of Their Impact on Performance*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[15] C. Huang, O. Lawlor, and L. V. Kale. Adaptive MPI. *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, 2003.

[16] 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.

[17] L. V. Kale, S. Kumar, J. DeSouza, M. Potnuru, and S. Bandhakavi. Faucets: Efficient resource allocation on the computational grid. *Proceedings of the 2004 International Conference on Parallel Processing (ICPP 2004)*, August 2004.

[18] L. V. Kalé and W. Shu. The Chare Kernel base language: Preliminary performance results. pages 118–121, August 1989.

[19] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[20] G. A. Koenig. An efficient implementation of Charm++ on Virtual Machine Interface. Master's thesis, Univeristy of Illinois at Urbana-Champaign, 2003.

[21] G. A. Koenig and W. Yurcik. Design of an economics-based software infrastructure for secure utility computing on supercomputing clusters. *Proceedings of 12th International Conference on Telecommunication Systems - Modeling and Analysis (ICTSM 2004)*, July 2004.

[22] V. Mehta. LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines. Master's thesis, Univeristy of Illinois at Urbana-Champaign, 2004.

[23] 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.

[24] V. K. Naik, S. K. Setia, and M. S. Squillante. Processor allocation in multiprogrammed distributed-memory parallel computer systems. *Journal of Parallel and Distributed Computing*, 1997.

[25] A. Natrajan, M. Crowley, N. Wilkins-Diehr, M. A. Humphrey, A. D. Fox, A. S. Grimshaw, and I. Charles L. Brooks. Studying protein folding on the grid: Experiences using CHARMM on NPACI resources under Legion. *Proceedings of 10th High Performance Distributed Computing*, August 2001.

[26] A. Natrajan, M. Humphrey, and A. Grimshaw. Capacity and capability computing using Legion. *Proceedings of 2001 International Conference on Computational Science*, May 2001.

[27] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurrency*, 5(2):60–73, April-June 1997.

[28] S. Pakin and A. Pant. VMI 2.0: A dynamically reconfigurable messaging layer for availability, usability, and management. Cambridge, Massachusetts, February 2002.

[29] A. Pant, S. Krishnamurthy, R. Pennington, M. Showerman, and Q. Liu. VMI: An efficient messaging library for heterogeneous cluster communication. http://www.ncsa.uiuc.edu/Divisions/CC/ntcluster/VMI/hpdc.pdf, 2000.

[30] G. F. Pfister. An introduction to the InfiniBand architecture. In H. Jin, T. Cortes, and R. Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. IEEE/Wiley Press, New York, 2001.

[31] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, January–March 1999.

[32] T. H. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. Ph.D. thesis, Computer Science, Graduate Division, University of California, Berkeley, CA, 1993.