Optimizing Distributed Application Performance Using Dynamic Grid Topology-Aware Load Balancing

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

Abstract

Grid computing offers a model for solving large-scale scientific problems by uniting computational resources owned by multiple organizations to form a single cohesive resource for the duration of individual jobs. Despite the appeal of using Grid computing to solve large problems, its use has been hindered by the challenges involved in developing applications that can run efficiently in Grid environments. One substantial obstacle to deploying Grid applications across geographically distributed resources is cross-site latency. While certain classes of applications, such as master-slave style or functional decomposition type applications, lend themselves well to running in Grid environments due to inherent latency tolerance, other classes of applications, such as tightly-coupled applications in which each processor regularly communicates with its neighboring processors, represent a significant challenge to deployment on Grids.

In this paper, we present a dynamic load balancing technique for Grid applications based on graph partitioning. This technique exploits knowledge of the topology of the Grid environment to partition the computation's communication graph in such a way as to reduce the volume of cross-site communication, thus improving the performance of tightly-coupled applications that are co-allocated across distributed resources. Our technique is particularly well suited to codes from disciplines like molecular dynamics or cosmology due to the non-uniform structure of communication in these types of applications. We evaluate the effectiveness of our technique when used to optimize the execution of a tightly-coupled classical molecular dynamics code called LeanMD deployed in a Grid environment.

1. Introduction

One of the attractive features of Grid computing [8, 9] is that resources in geographically distant places can be mobilized to meet computational needs as they arise. 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.

A particularly challenging issue when deploying Grid applications across geographically distributed computational resources is overcoming the effects of the latency between sites. While the interconnects used within today's clusters can typically deliver application-to-application latencies of a few microseconds, wide-area network latencies are usually measured in tens or hundreds of milliseconds. Certain classes of applications can achieve good performance in environments such as this. For example, applications that employ functional decomposition, such as climate models in which an atmosphere computation runs on one cluster and an ocean computation runs on another cluster, are very good candidates for deployment in Grid environments because the volume of communication traveling across cluster boundaries is much less than the volume of communication internal to each cluster. In contrast, some classes of applications present serious challenges to deployment in Grid computing environments. For example, tightly-coupled applications where each processor communicates with its neighboring processors during every iteration present a significant challenge. Coping with the effects of wide-area latency is critical for achieving good performance with these types of applications when deploying them in a Grid computing environment.

In previous work we have shown that it is possible to achieve good performance with tightly-coupled applications in Grid computing environments by leveraging latency tolerance features in an adaptive middleware layer [19]. By decomposing applications into a large number of message-driven objects, runtime systems such as Charm++ and Adaptive MPI allow time that would otherwise be wasted waiting for communication with neighbors across a cluster boundary to be overlapped with useful work driven by objects within the local cluster.

The contribution of this paper is the demonstration that the dynamic load balancing capabilities of the Charm++ and Adaptive MPI systems can be used to further improve performance for tightly-coupled applications running in Grid computing environments. The technique developed for this work exploits knowledge of the communication topology of the Grid environment to partition the computation's communication graph in such a way as to reduce the volume of cross-site communication, thus improving the performance of tightly-coupled applications that are co-allocated across distributed resources. In this paper, we focus our examination of the effectiveness of this technique on codes from disciplines like molecular dynamics or cosmology. In these types of problems, elements in the problem domain (e.g., atoms, planets, etc.) interact with other elements within a specified cutoff distance; elements outside this cutoff distance represent little or no influence on a given element. This characteristic presents unique opportunities for load balancing by mapping work to the processors in a Grid computation such that the amount of wide-area communication needed can be reduced.

2. Enabling Technologies

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

2.1. Charm++ and Adaptive MPI

Charm++ [14] is a message-driven parallel programming language based on C++ and designed with the goal of enhancing programmer productivity by providing a high-level abstraction of a parallel computation while at the same time delivering good performance. Programs written in Charm++ are decomposed into a number of cooperating objects called *chares*. Execution within chares is message-driven in a style similar to Charm++ contemporaries such as Active Messages [26], Fast Messages [22], and Nexus [10]. When a chare receives a message, the message triggers the execution of a corresponding method within the chare to handle the message asynchronously. Chares may be organized into indexed collections called *chare arrays*, and 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 mapped to processors by an adaptive runtime system. The mapping of chares to processors is transparent to the programmer, and this transparency permits the runtime system to dynamically change the assignment of chares to processors during program execution to support capabilities such as measurement-based load balancing, fault tolerance, automatic checkpointing, and the ability to shrink and expand the set of processors used by a parallel job. This idea is illustrated in Figure 1.

Adaptive MPI (AMPI) [13] provides the capabilities of Charm++ in a more traditional MPI programming model. AMPI implements the MPI standard by encapsulating each MPI process within a user-level migratable thread implemented as a Charm++ object. By embedding each thread within a chare, 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 the use of message-driven objects provides a high degree of latency tolerance to parallel applications. Because there are typically a large number of objects mapped to each physical processor in a Charm++ computation, time that would otherwise be wasted waiting for communication to take place can be spent performing useful work in ready objects. Our continued research into the efficient execution of tightly-coupled applications in Grid computing environments exploits this concept to a great degree by attempting to establish a favorable ratio of "border objects" (objects that communicate with neighbors across a cluster boundary) to "local objects" (objects that communicate with neighbors that are located entirely within the local cluster) on each processor in a Grid computation. By placing a small number of border objects relative to a large number of local objects on each processor, tightly-coupled parallel applications can be deployed successfully on a Grid with little or no impact to the application's performance.

A second enabling idea in Charm++ and AMPI that we directly leverage for the work described in this paper is the dynamic load balancing capabilities that result from the transparent mapping of objects to processors. The fundamental technique here follows from the observation that applications that employ a functional decomposition model (e.g., a global climate model consisting of discrete pieces such as an atmosphere model and an ocean model) usually exhibit good performance in Grid environments due to the fact that individual pieces communicate much less frequently with each other than they do internally. That is, the amount of wide-area communication is much less than the amount of local-area communication. By creating a Charm++ load balancer that has knowledge of the topology of a Grid environment in which an application is running, it may be possible to find a mapping of objects to processors in the computation in which the amount of wide-area communication is reduced. Applications that simulate physical

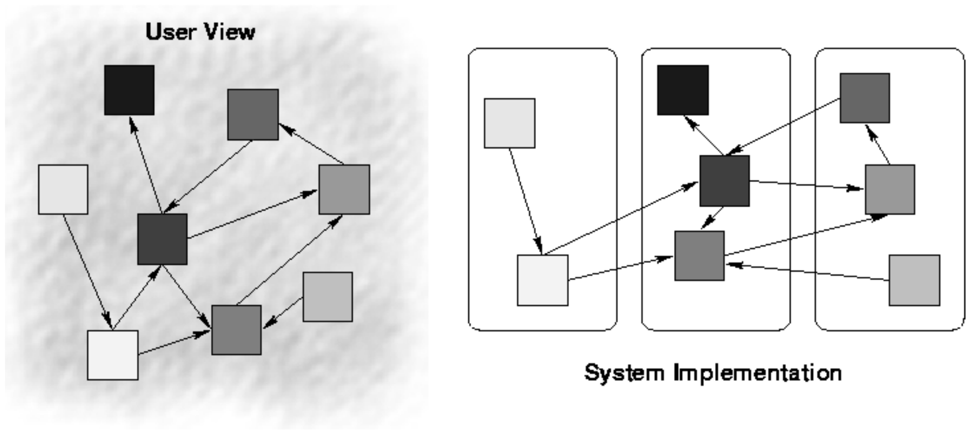


Figure 1. A depiction of the user's view of a Charm++ application and the system's view after mapping objects to processors

systems (e.g., classical molecular dynamics or cosmology) often present good opportunities for finding such mappings due to the non-uniform structure of communications that often is present in these types of problems.

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 [25]. The Virtual Machine Interface (VMI) message layer [24, 23] 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 [18]. This version of Charm++ is used for all work described in this paper.

3. Related Work

Our goal of efficiently executing tightly-coupled parallel applications in Grid computing environments is increasingly shared by other researchers as the capabilities and ease of access to Grid resources improve. Indeed, Grid-enabled applications for solving non-trivial scientific problems, such as Cactus-G [3], have been used for several years in environments such as the national TeraGrid [1]. Furthermore, Grid metacomputing systems like Legion [12] even share the use of Object Oriented Programming techniques to manage complexity in Grid applications with our work. Unlike systems such as Cactus-G that exploit application-specific characteristics to achieve good performance on a Grid, however, our work seeks to develop solutions at the runtime system level that can be applied to a wide variety of problem domains. And, unlike more general systems such as Legion which seem to focus on problems that can implicitly tolerate latency, such as parameter sweep applications [21], our work focuses directly on the problem of delivering good performance to tightly-coupled codes with a per-step time of tens or hundreds of milliseconds.

Efficient low-level message-passing capabilities that leverage the best network path connecting any pair of processes in a computation are extremely important for achieving good application performance in Grid computations. Our use of the Virtual Machine Interface provides similar capabilities as those found in systems such as MPICH-G2 [15] and MPICH/MADIII [4]. These projects, like ours, view the communication infrastructure of a distributed Grid job as a hierarchy of interconnects. MPICH/MADIII is particularly interesting because it uses an efficient user-level thread library, called Marcel, that provides task decomposition capabilities similar to what is available with Charm++ chares or AMPI threads, although MPICH/MADIII seems to typically use a much smaller number of threads per processor than is common with Charm++. Further, the Charm++ and AMPI runtime systems include 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. Indeed, the work presented in this paper would not be possible without this functionality.

Load balancing of parallel applications is a well-known concept with a history dating back more than twenty years. For example, Fox's book [11] describes load balancing through the use of randomized placement of sub-blocks within a problem. Interest in load balancing has increased in recent years in the field of Grid computing because the performance of Grid applications can be significantly improved through the use of good load balancing. Two primary ways of doing Grid load balancing exist: static techniques and dynamic techniques. When using static techniques, decomposed pieces of an overall problem are assigned to the most suitable (least loaded) processor in the computation. However, once a unit of work is placed on a processor, it remains on that processor until the work is completed; it cannot migrate to a new processor as in our system. We believe that static load balancing techniques cannot fully address the unique needs of Grid environments due to the constantly-changing nature of Grids. To that end, our work focuses on dynamic load balancing techniques provided by the Charm++ runtime system [27].

The dynamic load balancing capabilities of our work are similar to systems such as OptimalGrid [17] which monitor the runtime performance of each node in a computation with respect to the portion of a problem that it is handling and reapportion successive iterations of the computation to address load imbalances. A particularly in-depth analysis of this type of technique was carried out using a Successive Over-Relaxation (SOR) mesh problem running in the PlanetLab Grid environment [6]. The biggest difference between the dynamic load balancing research in OptimalGrid and in the PlanetLab experiments and our own work is that our work balances a computation primarily in terms of the object-to-object communication graph of the computation in relation to the structure of the Grid resources used by the computation and only secondarily on measured CPU utilization. In some ways, our work to efficiently balance Grid applications resembles other work within our research group for doing topology-aware load balancing within a single cluster [2], however our work generally focuses on finding ways to mask the effects of very high latencies found in Grid computations while the topology-aware task mapping work seeks to reduce contention for the high-performance low-latency networks within a single cluster.

The common thread that differentiates our work from others is our pervasive use of message-driven execution, in the form of Charm++ chares or AMPI threads, coupled with Grid topology-aware dynamic load balancing 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. Load Balancing of a Molecular Dynamics Application

In light of the enabling technologies described above, we now describe our design for a Grid topology-aware load balancer.

4.1. Grid Topology-Aware Load Balancer Design

A Grid computation can be thought of as representing a hierarchy of communication latencies. At the lowest level of this hierarchy is lightweight intra-processor communication, such as when two neighboring objects are co-located on the same processor. The next level of the hierarchy is represented by slightly heavier weight intra-cluster communication, such as when two neighboring objects are co-located within the same cluster. Finally, at the highest level of the hierarchy, heavy-weight cross-cluster communication is used when two neighboring objects must communicate with each other over high-latency wide-area communication channels. Latencies within this hierarchy can vary from sub-microsecond intra-processor latencies, to intra-cluster latencies measured in tens of microseconds, and finally to cross-cluster latencies measured in tens or hundreds of milliseconds. Efficiently mapping work to computational resources is a challenging problem even in traditional parallel applications running entirely within a single cluster that use only one interconnect. Performing such a mapping effectively in a Grid computation with multiple interconnects represents a significant challenge. This is further complicated by the fact that latencies in a Grid computation, particularly those for wide-area connections, may actually change as an application executes.

Dynamic load balancing like that used by systems such as Charm++ and Adaptive MPI presents several intriguing possibilities for efficiently executing applications in Grid environments. The system can observe the runtime behavior of the application, as well as the dynamic characteristics of the Grid environment itself, and make optimizations that may be non-obvious or difficult for humans to discover. An important point is that, in order to be most effective, this load balancing must take into consideration the communication hierarchy of the Grid environment itself. That is, the load balancing framework must be aware of the topology of the Grid environment in which an application is running.

The first step in creating a Grid topology-aware load balancer is actually discovering where the discrete breaks in the communication hierarchy are located. In the case of intra-processor latencies, this is trivial — the actual processors allocated to the computation define these boundaries. In the case of cross-cluster latencies, we need some way of identifying which processors belong to each cluster. We provide two solution to this problem. The simplest solution is to have the user specify this information at job submission time. We have implemented the capability for a user to launch the independent pieces of a single Grid job separately on their respective clusters. By augmenting this capability, Charm++ considers each sub-job launched as belonging to a unique cluster. A more complex solution is also available, and this solution has the advantage of automatically detecting the cluster to which each processor belongs. To do this, each processor measures the latency to every other processor in the computation when it opens connections during program startup and initialization. By collecting these latency vectors onto a single processor (by default, Rank 0 is used), it is possible to deduce the overall structure of the processors allocated to a computation. This information is then used to make topology-aware load balancing decisions.

After the topology of a Grid computation is discovered, the next step is to load balance the running application to reflect this topology. During an application's execution, the Charm++ load balancing framework collects statistics about the runtime characteristics of the objects in the application. Such statistics include the measured CPU load of each object as well as statistics about the number of messages and number of bytes sent between every pair of objects. Based on these statistics, an updated mapping of objects to processors can be computed by using graph partitioning techniques on the computation's object communication graph. This updated mapping hopefully reduces the volume of communication that crosses cluster boundaries, thus allowing the overall performance of the application to be improved. Graph partitioning is NP-complete. Fortunately, several software packages exist that can produce fast approximations to this problem. One such package is Metis [16], and we leverage this piece of software for our load balancing work. Our choice of Metis is based entirely on our familiarity with this software; any contemporary graph partitioning software could be used with similar results.

Simply partitioning the object communication graph into a number of partitions equal to the number of processors in the computation would not result in an optimal mapping, because this mapping would not reflect the fact that the inter-processor latency between some pairs of processors is much greater than the latency between other pairs in the Grid environment. Instead, we use a two-phase algorithm to partition objects to processors such that the cut of the object communication graph reduces the volume of communication across cluster boundaries. This algorithm is described as follows.

- **Phase 1:** In the first phase, objects are partitioned into the clusters in the Grid computation. At this stage, no consideration is given to balancing the computation on the measured CPU load of each object. Rather, the sole criteria for balancing is the measured number of object-to-object messages. This is to ensure that the partitioning of objects produced in this phase results in a cut in the communication graph that reduces the volume of communication crossing cluster boundaries.

  To carry out this phase, a graph is constructed for input to Metis that includes every object in the computation. Weights on the edges of the graph represent the number of messages passed between any pair of objects. Vertex weights are ignored. Metis is then instructed to partition the communication graph into a number of partitions related to the number of clusters in the computation. In some cases, the number of partitions used may not necessarily be exactly equal to the number of clusters in the computation, and this discrepancy is described below.

- **Phase 2:** In the second phase, objects within each cluster are partitioned onto the processors within their assigned clusters. This partitioning considers both the measured CPU load of each object as well as the object-to-object communication graph internal to each cluster, ensuring that the objects within each cluster are placed so as to produce both the most optimal CPU utilization on each processor as well as the most optimal intra-cluster communication graph. Inter-object communication that crosses cluster boundaries is ignored at this phase due to the fact that Phase 1 above has already partitioned the objects into clusters in such a way as to reduce the volume of communication on the edge cuts across cluster boundaries as long as each border object appears anywhere within the cluster to which it was assigned in Phase 1.

  To carry out this phase, graphs are constructed for input to Metis that includes every object in each cluster. Weights on the edges of these graphs represent the number of messages passed between objects. Vertex weights represent the measured CPU load of each object. Metis is then instructed to partition the graph into a number of partitions related to the number of processors in the associated cluster. Again, in some cases, the number of partitions used may not necessarily be exactly equal to the number of processors in a given cluster, and this discrepancy is described below.

After completing this two-phase process, the resulting object mapping ensures that all objects in the computation have been balanced in such a way as to produce a reduced volume of communication in the object communication graph across cluster boundaries as well as in a way that balances CPU load and intra-cluster object communication within each cluster.

As mentioned previously, the number of partitions requested from Metis in the phases of the algorithm described above may not necessarily exactly match the number of clusters in the computation (Phase 1) or the number of processors in a given cluster (Phase 2). This discrepancy is due to the possibility of a heterogeneous allocation of resources used in a Grid computation. For example, the processors within a single cluster may be of varying speeds. In such a case, it is desirable to allocate more work to the faster processors and less work to the slower processors. In order to get Metis to do this, the measured CPU speed of each processor, collected automatically by the Charm++ load balancing framework during program startup and initialization, is normalized against the slowest processor in each cluster. This produces a multiplier for each processor; the sum of these multipliers is used as the number of partitions for Metis. The resulting object map from Metis is then related to the physical processors in terms of this multiplier. That is, a processor that is twice as fast as the slowest processor in its cluster receives a multiplier of 2, and is accordingly assigned objects from two partitions of the object map produced by Metis. Similarly, and more likely in a real Grid computation, clusters may be of unequal power, due either to an unequal number of processors (e.g., Cluster A has twice as many processors as Cluster B) or to heterogeneous processor speeds between clusters. The solution here is to compute a multiplier for each cluster, based on the sum of the multipliers for the processors that make up each cluster, and to use the sum of these as the number of partitions for Metis.

4.2. LeanMD

To evaluate the effectiveness of our load balancing technique to the problem of efficiently executing tightly-coupled codes in Grid computing environments, we evaluate its use on a classical molecular dynamics code called LeanMD [20]. LeanMD is representative of scientific codes of reasonable complexity. Molecular dynamics codes typically employ a spatial decomposition style in which the atoms of a biomolecular system, composed of proteins, cell membranes, SNA, and waters, interact with the other atoms that are within a certain cutoff distance. Each timestep involves calculating the forces acting on all atoms and then using these forces to update the positions and velocities of each atom.

Within LeanMD, all computation takes place within a *simulation box*, a bounding box for all atoms involved in the simulation. The space within a simulation box is divided into regular cubic regions of simulation space called *cells*. Each cell is responsible for all the atoms that fall within its boundary, their coordinates, and the forces exerted on them. As described above, molecular dynamics simulations involve the interaction of atoms with other atoms within a certain cutoff distance. In a simulation with a k-away cutoff distance, a cubic region of simulation space formed by $k \times k \times k$ cells is considered. This cubic region of space is called a *patch*.

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 timestep, each cell "integrates" all forces on its atoms and changes their positions based on new acceleration and velocities calculated. It then multicasts its atoms' coordinates to the 26 cell-pairs ($3 \times 3 \times 3$ cube) that depend on it.

The molecular system considered here consists of approximately 30,000 atoms and 3,000 cell-pair-objects. Thus, each processor in the computation contains several tens of cell-pair objects. In a multi-cluster context, some subset of these objects ("subset A") require messages from cells within their own cluster, while a different subset ("subset B") may require interaction with cell-pair objects from outside the cluster. As a result of the message-driven model of execution employed by Charm++, 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 as shown in our previous work. The technique used in this paper further optimizes the application by attempting to identify objects in subset B in which the number of interactions with remote objects exceeds the number of interactions with local objects, and migrating (load balancing) these objects so that they are nearer to their neighbors with which they communicate most, thus reducing the overall volume of cross-cluster communication necessary in each timestep.

5. Experimental Results

In this section, we describe a set of experiments based on the LeanMD molecular dynamics application described in Section 4. Results of these experiments demonstrate that our load balancing technique can be used to significantly improve the performance of applications in Grid computing environments.

5.1. Experimental Environment

All experiments described in this paper are carried out in a simulated Grid computing environment consisting of a pair of clusters. All cluster nodes in this environment 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., 16, 32, 64, and 128 processors) and these processors are evenly distributed between the two clusters (i.e., 8+8, 16+16, 32+32, and 64+64 processors). Nothing about the underlying problem requires a power of two processors. In fact, a fundamental characteristic of Charm++ and Adaptive MPI is that the number of objects used to decompose the problem is independent of the number of processors. However, choosing a power of two processors for our experiments allows us to ensure an even division of processors between cluster boundaries. That is, 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.

In practice, Grid computations that span multiple clusters generally run on environments consisting of two to four independent clusters. We believe that conducting the experiments described in this paper on two clusters is reasonably representative of what might be experienced in a real Grid computation. More importantly, our decision to use two clusters allows us to reason better about the impacts of latency on the application due to the fact that the communication hierarchy consists of only two levels, a low-latency level within the two clusters and a high-latency level between the two clusters. A Grid environment consisting of three or more independent clusters would make the task of understanding our results more difficult and we do not believe would greatly enhance the conclusions that can be drawn.

A simulated Grid environment is constructed using nodes that physically exist within a single real cluster. 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 predefined 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 predefined amount of time before passing it to the network device driver used to communicate over the "wide area." Our previous work [19] has shown that the performance of jobs running in this simulated Grid environment closely matches the performance of jobs running in real Grid environments.

5.2. Load Balancing Results

We evaluated the effectiveness of our load balancing technique when applied to LeanMD by conducting a series of experiments that show the effects of increasing latency on the application's per-step performance. Four main experiments were run, each with a fixed number of processors. For each of these experimental runs, 12 data points were generated corresponding to increasing values of artificial latency. For each fixed processor size and latency 2,000 iterations of LeanMD were run with a single load balancing operation performed after 1,000 iterations. Our choice of 2,000 iterations with load balancing after 1,000 iterations was somewhat arbitrary. LeanMD allows the user to configure the simulation to run for a fixed number of iterations with load balancing taking place at user-specified intervals. It is expected that for real computational science conducted with LeanMD, the user will select these values based on their experience with the molecular system being studied. For our purposes, running 2,000 iterations with load balancing after 1,000 iterations allows LeanMD enough time to reach a stable state in terms of the per-iteration time after startup and after load balancing, thus allowing us to discard several iterations at the beginning of the run that include startup costs and several iterations at the middle of the run that include load balancing costs.

The median per-step time of the application in the first 1,000 iterations (i.e., without load balancing) was compared to the median per-step time of the application in the second 1,000 iterations (i.e., after load balancing) to determine the effectiveness of load balancing on the application's performance. Figure 2 shows the per-step execution time of LeanMD as a function of cross-site latency on different numbers of processors. Each graph shows results without load balancing and with our graph partitioning load balancer (denoted "GridMetisLB"). As a point of reference, results for a second load balancer (denoted "GreedyLB") are also presented. This load balancer uses a greedy algorithm that adjusts the object mapping based only on the measured CPU load of each object, ensuring that each processor in the computation has roughly an equal CPU load. GreedyLB does not consider the communication characteristics of the computation in any way. Thus, comparisons with these results provide a reasonable idea of the effectiveness of our Grid topology-aware load balancing technique.

In the graphs, horizontal lines are desirable because they show that the per-step execution time of the application does not increase with increasing latency. For 16 processors (Figure 2(a)), for example, the natural parallelism of the application along with the relatively large per-step time (1,000 milliseconds for the unbalanced case) completely mask the effects of latency due to the ability of Charm++ to overlap wide-area communication with locally-driven work as described in our previous paper [19]. Both GridMetisLB and GreedyLB produce similar improvements in performance, and these improvements are due entirely to both balancers adjusting the object mapping in terms of each object's measured CPU load.

The results for 32 processors (Figure 2(b)) show similar trends. Doubling the number of processors gives good scaling of the application even in the unbalanced case. The results for both load balancers improve the per-step execution time from approximately 550 milliseconds per step to approximately 450 milliseconds per step. The results for GridMetisLB, however, are slightly better than those for GreedyLB through 80 milliseconds of cross-site latency. These results suggest that the graph partitioning technique employed by GridMetisLB produces a more optimal object mapping compared to GreedyLB which balances the computation only in terms of each object's measured CPU load. More interestingly, the results for the load balanced cases between 80 and 128 milliseconds show an increase in the per-step execution time for the GreedyLB case while the GridMetisLB case remains flat. This further underscores the effectiveness of our graph partitioning load balancing technique.

For 64 processors (Figure 2(c)), the unbalanced case shows poor scalability with the doubling of processors, and this result is known to be due to CPU load imbalance [20]. The results for both load balancers significantly improve application performance, although again the results for GridMetisLB for low cross-site latencies are slightly better than those for GreedyLB. As the cross-site latency increases, the results for both load balancers show an increase in the per-step execution time, indicating that the effects of latency cannot be masked entirely. However, the results for GridMetisLB remain horizontal through a longer range of cross-site latency, through the neighborhood of 48 to 64 milliseconds, compared to the results for GreedyLB. Further, in the region of the graph when the results for GridMetisLB show an increase in execution time as latency increases, from 64 to 128 milliseconds of latency, the results for our graph partitioning load balancer are better than the results for CPU load balancing alone.

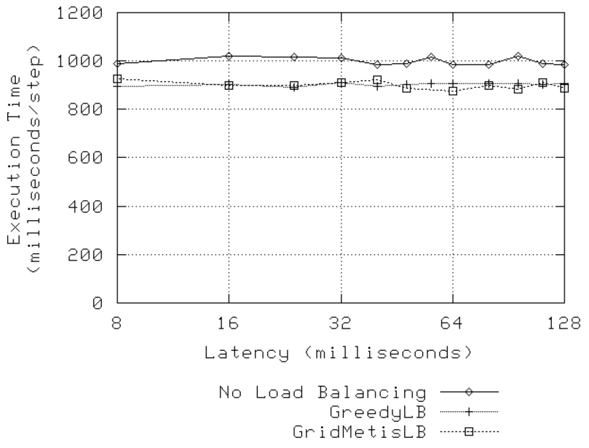
In the 128 processor graph (Figure 2(d)), the results for all three cases show an increase in the per-step execution time as cross-site latency increases. Again, not only do the results for GridMetisLB remain horizontal longer than the results for GreedyLB (through 24 milliseconds latency), the results for GridMetisLB are consistently better than those for GreedyLB through 128 milliseconds latency. This strongly suggests that the object mapping produced by our graph partitioning load balancing technique is more favorable for optimizing performance of the application in Grid computing environments than the object mapping produced by considering measured CPU load alone.

Load balancing itself incurs some amount of overhead. The cost of instrumenting the Charm++ runtime system itself is measured to be minimal [27]. We further measured the cost of GridMetisLB as approximately 500 milliseconds for each invocation and believe this is reasonable when considering that load balancing in real-world applications is likely to be carried out only once every several minutes of execution.

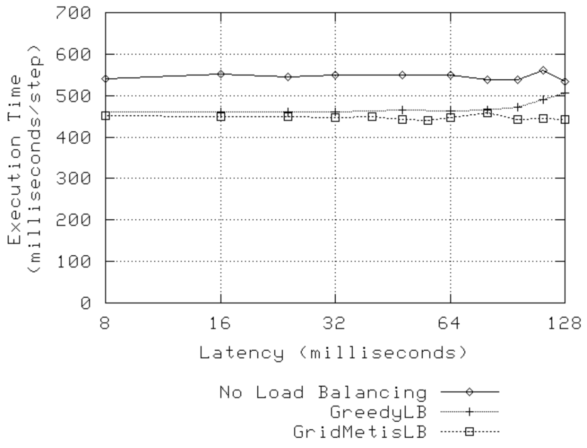
6. Conclusion

In this paper, we have demonstrated that the use of Grid topology-aware dynamic load balancing in Charm++ can be used to optimize tightly-coupled distributed applications running in Grid computing environments. The performance benefits gained by using our load balancing technique on these codes improve upon the performance benefits related to the use of message-driven objects for masking wide-area latencies, as reported in our earlier work.

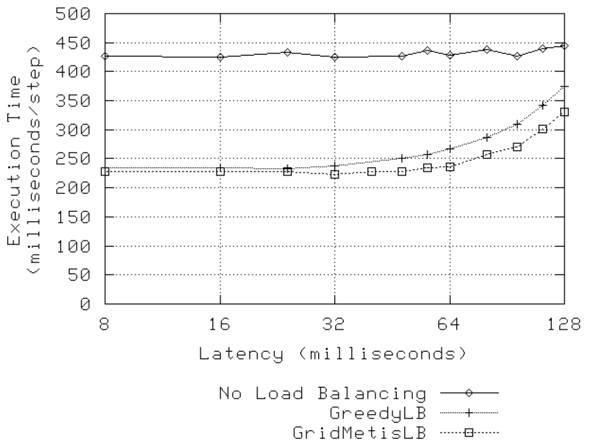
In addition to the results described in this paper, we have developed Grid load balancing techniques that can be applied to applications in other problem domains such as regular and irregular mesh decomposition. These types of applications present challenges that are distinct from the challenges of "particles in boxes" applications because the communication patterns in mesh applications are frequently very uniform. That is, it is generally not possible to partition the application in any way that significantly reduces the volume of wide-area communication. For these cases, we employ a strategy of simply distributing the objects that communicate across high-latency wide-area connections evenly



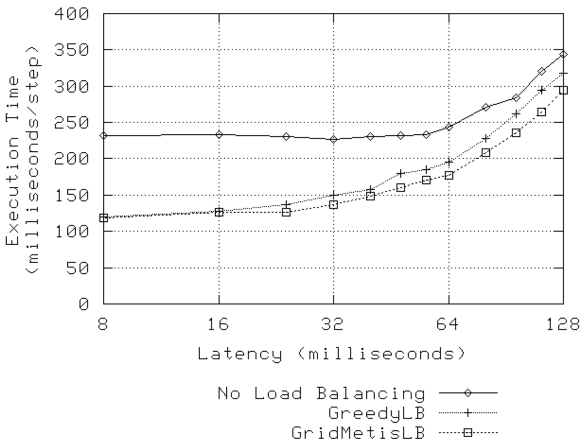
(a) 16 Processors



(b) 32 Processors



(c) 64 Processors



(d) 128 Processors

Figure 2. Performance of LeanMD with artificial latencies 1-128 milliseconds

among the processors within a cluster. In this scheme, no objects are migrated to remote clusters; rather they are simply migrated among the processors within the cluster in which they were originally placed. In this way, a favorable ratio of border objects to local objects can be established on each processor, providing the the most possibilities for overlapping the wide-area communication in border objects with locally-driven work.

References

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

[2] T. Agarwal, A. Sharma, and L. V. Kale. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[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] M. Dobber, G. Koole, and R. van der Mei. Dynamic load balancing experiments in a grid. *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2005.

[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. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *SIAM Journal on Scientific Computing*, volume 20, pages 359–392, 1998.

[17] J. H. Kaufman, G. Deen, T. J. Lehman, and J. Thomas. Grid computing made simple. *The Industrial Physicist*, August/September 2003.

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

[19] G. A. Koenig and L. V. Kale. Using message-driven objects to mask latency in grid computing applications. In *19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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