An Adaptive Job Scheduler for Timeshared Parallel Machines

Laxmikant V. Kalé
Sameer Kumar
Jayant DeSouza
Department of Computer Science
University of Illinois at Urbana-Champaign
{kale,skumar2,jdesouza}@cs.uiuc.edu

Abstract

Computational power, at least at the high end, can be thought of as a utility, similar to electricity or water. To make this metaphor work requires a sophisticated "power distribution" infrastructure. The "Grid", popularized by the Globus project, is an example of such an infrastructure. To function efficiently, the producers of compute Power — the parallel servers — must be able to reorganize their jobs dynamically so as to respond to demands for computational power quickly, and maximize their utility. We are developing a framework, called *faucets*, that aims at facilitating this process. This paper focuses on a system at the heart of this framework: an adaptive manager for timeshared parallel machines that can shrink and expand its jobs to a variable number of processors dynamically. This manager has been implemented for workstation clusters. The paper describes the *faucets* framework, the design of the adaptive job manager, and preliminary performance data.

1 Introduction

Due to advances in hardware, desktop computers, including low-end parallel machines, can be used to solve many computational problems within required response times. Yet, there are many other computationally intensive applications that require much larger parallel machines. These applications tend to be in scientific and engineering simulations, as well as in emerging areas such as operations research, data-mining, decision support and corporate planning. Large parallel computers, shared by multiple users, are commonly used for running such applications. We expect this usage pattern to continue because of the economies of scale, advantages of specialization, and advantages of pooling of shared resources that such an arrangement yields. For example, an application of a single user may require the memory

of a 1000 processor machine to fit the simulation model, yet the user may need to run the program only for a few days per month, on the average, on such a machine. Such a user may not be able to afford to buy and maintain a single large machine. Also, users of a parallel application are not necessarily experts in maintaining a parallel machine. So, the trend towards specialization argues for having separate organizations for managing parallel computers.

The scenario that we envisage for the future is one in which computational power is "produced" in much the same manner as other utilities such as power and water are produced. Consumers of this power will have access to variable amounts of power on demand. This scenario is propounded and explored by the "Grid" project[6], and the associated frameworks such as Globus [7].

In this scenario, the users will submit jobs from their desktop computers, typically via web browsers. Users will have certain quality-of-service requirements, such as the total memory requirement, and deadlines. The "grid" will run the job on some available parallel computer, uploading files to it, and downloading results back to the desktop, when necessary. The parallel computers themselves will be run by for-profit centers which will charge for the compute power used by applications in some fashion. [1]

To make this scenario work effectively, some technical obstacles must be overcome. Challenging issues such as authentication, security, the maintenance of a directory of available servers, and the ability to run a single job on multiple parallel computers (which may be necessary for some of the extremely high-end applications) are being handled by other grid-oriented projects such as Globus[7]. Such projects have already developed infrastructure that supports commonly needed tasks such as uploading and downloading of files from desktops to parallel computers. However, some additional issues need further research.

From the point of view of the server, it is desirable for it to be able to implement strategies that will maximize its "profit" (or *utility* in case of in-house servers). In the current technology, the options for a server are limited: it can manage its queue of waiting jobs in some priority order, and sometimes it can checkpoint (or terminate, and restart later) an already running job in favor of a high priority job. As we will show in the paper, these options are not adequate to optimize its utility metric.

We describe an adaptive job scheduler system that allows servers to shrink and expand existing jobs to different numbers of processors at runtime, thus providing the needed flexibility for managing the parallel server. Although prior systems, such as DRMS [9], have aimed at such an ability, we will show how our scheme is more general, more widely applicable, and potentially more efficient.

In the next section we describe the underlying Charm++ framework, and especially its load balancing capabilities, that enables our adaptive job scheduler. In section 3 we describe the design and implementation of the job scheduler. Section 4 discusses the advantages of the

[1] One can then imagine that organizations such as NSF will pay the centers indirectly, by funding application scientists for the compute power they plan to use. This may bring about additional efficiencies in private management of such centers.

scheduler capable of shrinking and expanding running jobs. Performance data demonstrating the efficiency of the implemented system is presented in section 5. Section 6 compares our approach to other relevant systems implemented in the past. Section 7 concludes with a summary and future work.

2 The Underlying Framework: Charm

Since the Charm load balancing framework is at the heart of our approach, we describe it next.

The Charm framework supports multithreading and dynamic load balancing for applications written in MPI, Charm++, as well as other experimental languages. The framework is based on the Converse run-time library, which among other things, supports multithreading and portability across a wide variety of parallel machines including workstation clusters.

The load balancing capabilities are based on multi-partition decomposition (which is similar to virtualization). The user program breaks down the parallel applications into a large number of medium-grained pieces, or *chunks*. The number of chunks is chosen to be independent of the number of processors, based solely on the need to amortize communication overhead. Thus, one chooses the smallest possible grain size that is adequate. This typically leads to the number of chunks being much larger than the number of processors. In an MPI program, the user specifies a large number of MPI processes, which are implemented as user-level threads by AMPI [1], our adaptive implementation of MPI. In Charm++, the program consists of a large number of communicating objects. Both AMPI and Charm++ allow and support migration of their chunks (threads and objects, respectively) from processor to processor.

The load balancing system automatically instruments the run-time system, so as to monitor the amount of time spent in each user level entity. It also monitors the communication pattern between these entities. (The instrumentation overhead is small, and can be turned on for a time window just before load balancing). Periodically, or on demand, it uses the collected statistics to remap the chunks to processors, achieving better load balance, and possibly reduced communication overhead.

The efficacy of the Charm load balancing framework has been demonstrated in several contexts. For example, NAMD, a parallel molecular dynamics program used routinely by biophysicists, uses this framework to achieve scalability. It has achieved unprecedented speedups for this application (1,252 on 2k processors), leading to a Gordon Bell award entry (finalist at SC2000) [3]. We have also used this framework to deal with external interference [4] on parallel jobs running on clusters: specifically, the system detects and reacts quickly to ameliorate performance degradation caused when another user logs on to (and starts a computation on) one of the workstations being used by a parallel job.

The capabilities of this load balancing framework provided the initial idea, as well as the implementation mechanism, for the adaptive job scheduler, which is described next.

3 The Adaptive Job Scheduler

The job scheduler engages in two separate tasks: it participates in quality-of-service contract bids with outside users, and it enqueues and schedules the jobs it has committed to. It is the second task that we focus on in this paper. Thus, we assume that before accepting a job, the scheduler has made sure that it can meet the quality of service requirements.

At the minimum, the job's QoS requirements include the minimum and maximum number of processors the job can run on, an estimated number of CPU-seconds required by it, and a deadline before which it must be completed. It may also include such information as per-processor maximum memory requirements, and estimated parallel efficiency.

Given this, the objective of the scheduler is simply to maximize its *utility metric* while ensuring that the quality of service requirements are met. For the relatively simple QoS contracts described above, the utility metric is simply the overall processor utilization. (More complex contracts may specify a differential weighing system that rewards faster completion of jobs, for example.)

The scheduling decisions are taken at the following *control points*.

1. A new job arrives.
2. A running job terminates.
3. An alarm set by the scheduler itself is activated.

At each one of these control points, the scheduler may decide to initiate one or more of the following actions: 1) Start a new job. 2) Checkpoint and suspend a running job. 3) Shrink a running job to a smaller number of specified processors. 4) Expand a running job to a larger number of specified processors.

Most current job schedulers use only the first two actions while scheduling jobs.[2] The ability to shrink and expand jobs is a great asset to our job scheduler. We now describe the implementation of our system.

3.1 Implementation

The Charm++/Converse runtime system provides object migration and load balancing. The shrink/expand mechanism is implemented on top of this framework, so the parallel jobs are expected to be written in Charm++, MPI[1], or other languages ported to Charm++.

As shown in Figure 1, the system has three major components:*the scheduler* to schedule the parallel jobs, *the runtime support* which helps in migrating the load, and *the job-submission client* to remotely submit the jobs and monitor them.

[2]The checkpoint system we plan to use is more sophisticated than traditional ones, and allows the checkpointed job to be restarted on a different number of processors [11].

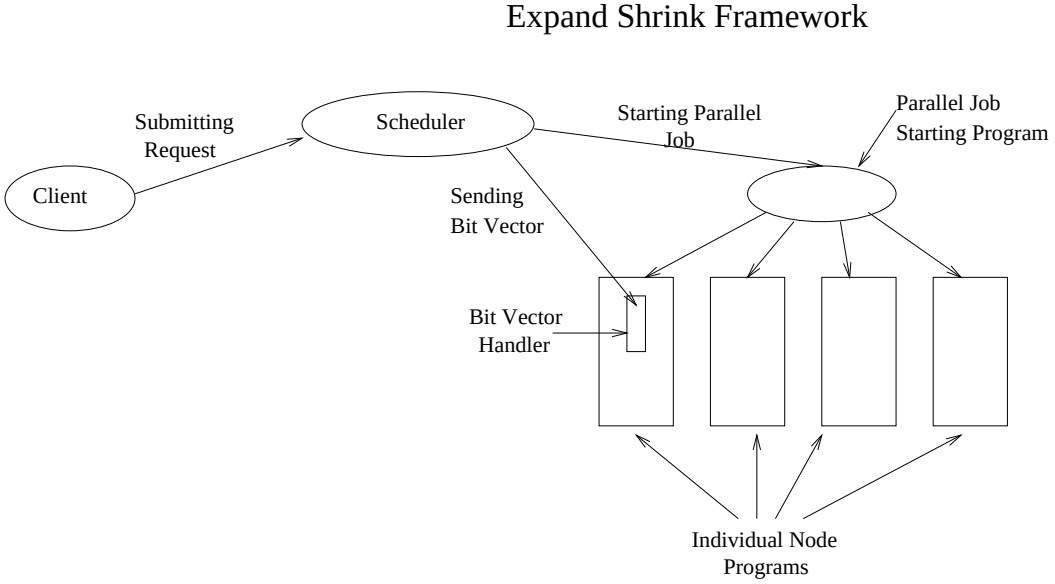

Figure 1: System Components.

The scheduler (one per workstation cluster) runs as a server listening on a well-known port. A client connects to it and requests the execution of a job. After some negotiation, the scheduler might accept the job. It then schedules the job using the following algorithm.

The scheduler maintains a table to keep track of which job is currently assigned to each PE. Let Job_n be a new job, with parameters $priority_n$, $minPE_n$, and $maxPE_n$.

1. If the number of free PE's $\geq maxPE_n$, we give the job all $maxPE_n$ PE's.
2. Otherwise, we determine whether we can give Job_n the desired $maxPE_n$ by shrinking lower priority jobs.
3. Otherwise, we try suspending lower priority jobs.
4. Otherwise, we perform Step 2 and then Step 3 trying to obtain at least $minPE_n$ PE's.
5. Otherwise, we cannot give Job_n even $minPE_n$, so we enqueue it.
6. Increase the priority of queued jobs to avoid starvation and raise alarms if necessary.

The scheduler starts the job and informs it which processors to use by passing it a bit-vector, of length the maximum number of processors in the system. All bits representing the processors allocated to the job are set to 1, and others are cleared to zero. Subsequently, the scheduler communicates with the running job via the *CCS* (Converse Client Server) protocol, which is a mechanism provided by the Converse system. To ask a parallel job to shrink or expand, the scheduler sends it (via CCS) a similar bit vector with the new processors set.

The bit-vector handler. When a job receives a bit-vector from the scheduler via CCS, the bit-vector handler is invoked. The handler sets the bit-vector in the load balancer. At the next load-balancing cycle, the load balancer will use the newly set bit vector for the job.

The load-balancer.

We have modified the load-balancer described in Section 2 to use the bit-vectors described above to expand and shrink jobs. The advantage of this is that parallel job remains load-balanced. But the shrink and expand mechanism has a latency of one load-balancing cycle. In future implementations the handler will migrate objects out of a deallocated processor immediately after being invoked.

The object manager. The object manager is a Charm++ component that performs object migration. The load balancer asks the object manager to migrate the objects to the PE's represented by the bit vector. During the process of migration the object manager buffers any messages to the migrating object.

A *r*esidual process is left on emptied processors. The overhead of this process is very small as shown in Section 5. The load consists of a transient period of forwarding messages, and periodic (but nominal) participation in global operations such as reductions and load balancing.

4 Discussion

We believe that the Shrink/Expand capability makes the *Adaptive Job Scheduler* more flexible, thereby providing several benefits. The key fact is that nodes that would be unused on a traditional queuing system (TQS) can be used by expanding one or more jobs. Further, a fully occupied cluster can be squeezed a little bit to fit in an important job. A TQS could achieve this only by suspending an existing job, which might have the undesired side-effect of freeing more nodes than are needed.

- An *Adaptive Job Scheduler* can utilize the system better while responding to higher priority jobs. In a typical queueing system, a higher priority job will preempt a lower priority job. With the *Adaptive Job Scheduler*, the lower priority job is resized to run on fewer CPU's, if possible.

- An *Adaptive Job Scheduler* can improve system utilization.

  With traditional jobs, the fixed number of nodes needed can sometimes lead to wasted nodes if two jobs together need more nodes than are available. The *Adaptive Job Scheduler* can sometimes avoid such waste. For example, consider two jobs J1 and J2 on a 128-node cluster. If J1 needs 96 nodes and J2 requires 64 nodes, then nodes will be wasted (unless nodes are time-shared). If, however, these were *Shrink/Expand Job*, then if J1 could run on, say, 80+ nodes, and J2 could use 40+ nodes; they could both run, and when one completed, the other would grow to use more CPU's, thereby improving system utilization.

- Idle-cycles jobs. One can now conceive of running a job just to use idle cycles. This job would have a very low priority, and would expand whenever higher-priority jobs were not using some nodes, (e.g. one can run a SETI screen-saver for clusters!)

- Usually, parallel programs have lower efficiency when run on more nodes. In such cases, a side-effect of shrinking down a job is that its efficiency improves. Thus running multiple jobs on the same cluster using time-sharing or FIFO might reduce throughput when compared with running the same jobs using an *Adaptive Job Scheduler*. For example, J1 and J2, both needing from 32 to 64 nodes, can be run FIFO on a 64-node cluster; or they can be run using an *Adaptive Job Scheduler*, which might allocate 32 nodes to each. If the jobs run at higher efficiency on 32 nodes, the completion time of both the jobs together will go down compared to the first case.

  However, the average response time of the jobs might worsen (go up). e.g. J1 and J2 take 120 time units on 32 PE's, and 75 on 64 PE's. The average response time is:

  - using FIFO: $((75-0)+(150-0))/2 = 112.5$
  - assigning both equally: 120

  The compute-provider might therefore give a discount to jobs with lower response time demands.

  One must mention, however, that there are cases in which average response time actually improves (goes down). e.g. J1 and J2 both take 100 time units each when run on 32 PE's, and 75 time units when run on 64 PE's. It they both start at the same time on an empty 64 PE cluster, the average response time is:

  - using FIFO: $((75-0)+(150-0))/2 = 112.5$
  - assigning both equally: 100

  Since our algorithm specified above does not shrink equal priority jobs, we will not obtain the benefits of this case; however, we will not under-perform FIFO with preemption.

- An *Adaptive Job Scheduler* could respond better to small-duration jobs.

  Currently, small jobs have to wait their turn. The wait may take a long time if a big job is running. The alternative it to suspend the big job in favor of the small one. One disadvantage of this is the overhead of suspending and restarting a big job; and the other disadvantage is that the small job may not need all the nodes, which means nodes will be wasted.

  With the *Adaptive Job Scheduler*, the small job can be given a chance to run by shrinking the running big job, and expanding it again when the small job completes. This avoids the overhead of swapping out the big job; and it can potentially better utilize the nodes.

5 Experimental Results

We conducted experiments on our scheduler, by running a molecular dynamics program similar to [2], and we obtained the following average times for shrinking and expanding of a single job. The job was a molecular dynamics simulation which simulated over 400,000 molecules. The total memory requirement of the job was about 44.8 MB. We ran the job on the Turing Linux Cluster [12] and obtained the following times for job shrinking and expanding.

Processors before and after shrink	Time on ethernet, s	Time on Myrinet, s
64 to 32	2.802	N.A.
32 to 16	1.315	.330
16 to 8	1.164	.362
8 to 4	1.477	.435

Table 1: Shrink Time for MD Program (400K atoms, 44.8 MB)

Since the cluster uses Ethernet all the data transfer is serialized, hence the transfer time first decreases due to parallel handling of messages and then increases due to collision between nodes.

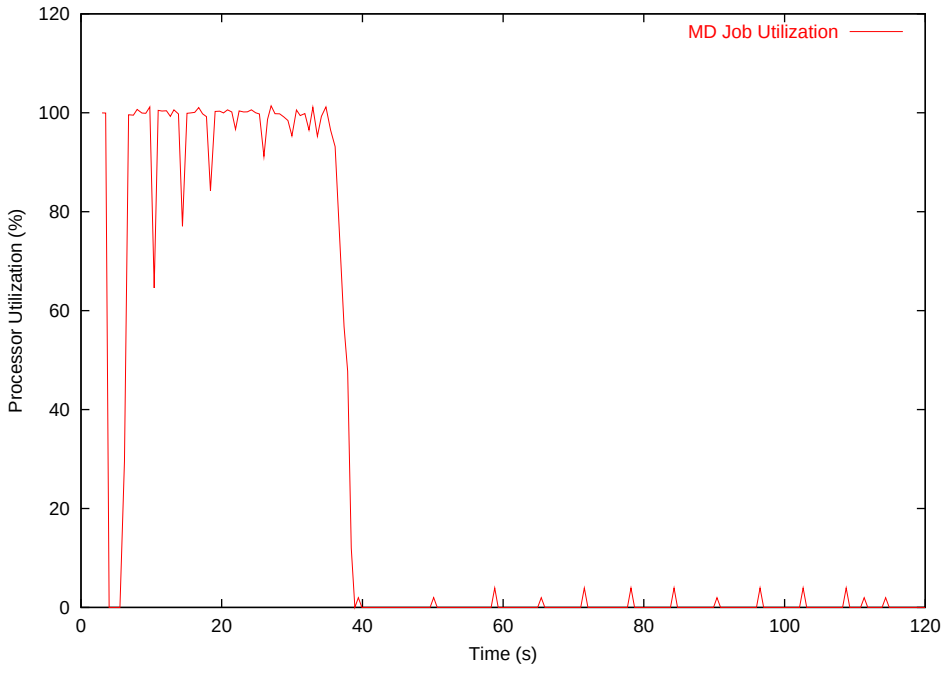

Figure 2: Residual Load after Shrinking.

When shrunk, the job leaves a residual load on the processors it vacates as described in Section 3.1. Figure 2 shows the load on one of the processors after the job has been vacated. This load is zero most of the time but has periodic peaks of 1.96 percent. This will cause other applications negligible interference. Work is being done to avoid this.

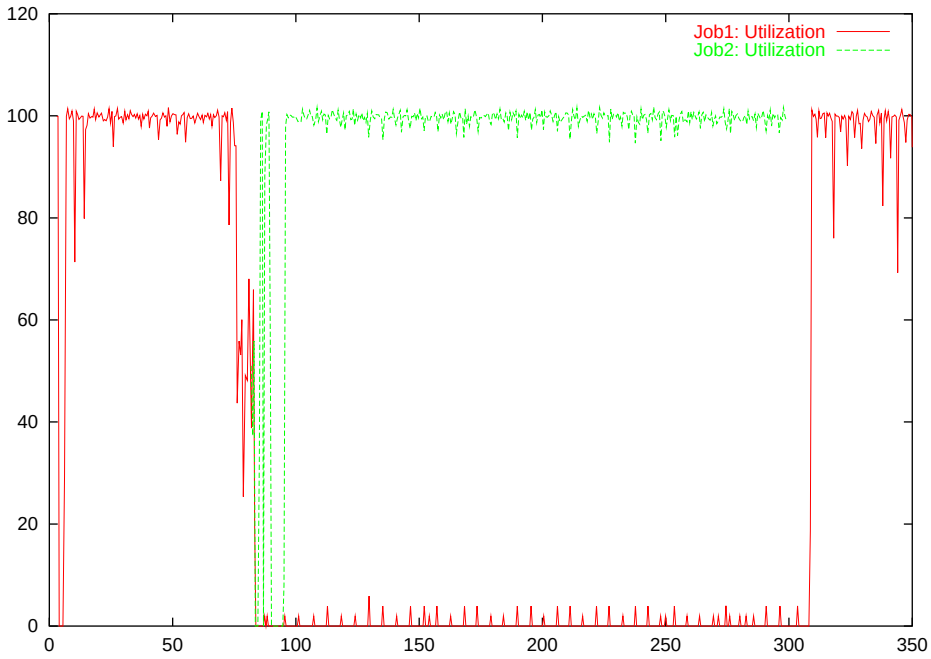

Figure 3: Proof of Concept of Shrink/Expand.

Figure 3 shows the shrinking and expanding of Job1 directed by the arrival and completion of Job2. On arrival of Job2, Job1 first start to share the load and at the next load-balancing cycle it is shrunk. When Job2 finishes, SIGCLD is sent to the scheduler by the operating system, and the scheduler asks Job1 to expand. Despite the interference by Job1, Job2 still gets most of the CPU, as shown in Figure 3.

6 Related Work

In this section we present other efforts in dynamic processor allocation to applications.

The initial efforts towards dynamic processor reconfiguration were mainly in the direction of dynamic process migration and load balancing. Condor [8] supports runtime migration of a process from one workstation to another, through checkpointing. This system only migrates sequential programs executing on one processor. Condor also supports the concept of negotiation, where each client bids for certain units of time.

Dome [10] is an object oriented distributed framework where applications are loadbalanced by migrating objects from heavily loaded processors to lightly loaded ones. Dome

does not support the concept of shrink and expand.

The ability to shrink and expand the number of processors allocated to a job was implemented by AMP (Adaptive Multiblock PARTI) framework [5]. This framework runs SPMD Fortran programs. Each processor that the program is started on can run an active process or a skeleton process. The active processes have all the load and the skeleton processes run in the background. To expand, the load is moved from the active to the skeleton tasks. The maximum number of parallel processes is the number of processors the program is started with. During a shrink or an expand the whole data is redistributed to the new set of active processors and the communication schedules are updated. The data redistributed is mainly the shared arrays.

DRMS [9] is another system that provides SPMD programs to reconfigure themselves dynamically. It is built on top of the SOP (schedulable and observable points) model. In this model the parallel program is examined at these SOP's and modified. If the number of tasks in the program is changed after the SOP, then it is a reconfiguration point. On reconfiguration all the global data is redistributed. The global data is an array of basic datatypes which is ordered as block, block-cyclic etc. Redistribution can be a very long process for irregular data processor mapping, so programs can specify a valid list of the permitted number of processors.

The DRMS scheduler as described in [13] categorizes jobs as small, medium or large (depending on the execution time they take). Large and medium jobs are allowed to be reconfigured while short jobs are not. The scheduling schemes try to minimize the average response times of the jobs and also the response time variance. Care is taken to prevent starvation of small jobs.

The redistribution of data, divides the data evenly among all the processors in use, but this does not imply that all the processors being used by the job are loadbalanced. Moreover no job priorities are considered.

Our framework differs from these projects in its broader applicability, (e.g. beyond data-parallel programs,) and its use of measurement-based load balancing, which is more accurate than even redistribution of data.

7 Summary and Future Work

We described the motivation, design, and implementation of an adaptive job scheduler for parallel machines. The scheduler builds upon a measurement based load balancer, and can be used by applications written in a variety of languages including MPI and Charm++. The original load balancers, which aimed at resolving application induced load imbalances, were extended to shrink, expand or to change the sets of processors allocated to a job. The load balancing strategies accomplish this by migrating user level entities (such as MPI threads and Charm++ objects) across processors. Mechanisms for facilitating socket based communication between a parallel job and outside processes, as well as for controlling the behavior

of load balancers via bit vectors of available processors were implemented and validated. We also described a simple job scheduling strategy, and some preliminary performance data.

The system described is a part of a wider *faucets* project, which aims at supporting the metaphor of computing power as a utility. Our future work will include expanded notions of quality-of -service contracts, and correspondingly sophisticated scheduling strategies that attempt to optimize more complex utility metrics than just processor utilization. The current system has been tested on clusters of workstations. We plan to port and evaluate the system on dedicated parallel machines, such as IBM SPx, which allows socket based communication with outside processes. We expect such a port to be straightforward (to be completed before the final version of the paper). Integrating the job scheduler with an automatic check pointing system, rather than relying on application-specific check pointing, is another direction for future work.

The faucets framework will include sophisticated quality-of-service contracts, competitive bidding processes, web based submission, monitoring and interaction with parallel jobs, along with services such as authentication, and file upload and downloads. We plan to utilize the common components in the Globus framework, and interoperate with Globus, so that the job scheduler runs as a Globus server.

References

[1] Milind Bhandarkar, L.V.Kalé, Eric de Sturler, and Jay Hoeflinger. Object-based adaptive load balancing for mpi programs. Technical Report 00-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Sept. 2000. Submitted for publication.

[2] John A. Board, L. V. Kalé, Klaus Schulten, Robert Skeel, and Tamar Schlick. Modeling biomolecules: Larger scales, longer durations. *IEEE Computational Science and Engineering*, 1(4), 1994.

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

[4] Robert K. Brunner and Laxmikant V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.

[5] G. Edjlali, G. Agrawal, A. Sussman, and J.Saltz. Data parallel programming in an adaptive environment. In *Proceedings of the 9th International Parallel Processing Symposium*, 1995.

[6] I. Foster and C. Kesselman (Eds). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[7] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[8] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Usenix Winter Conference*, 1992.

[9] 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.

[10] J. Nagib, C. Árebe, A. Beguelin, and Bruce Lowekamp. Dome: Parallel programming in a distributed computing environment. In *Proceedings of the International Parallel Processing Symposium*, 1996.

[11] Sameer Paranjpye. A checkpoint and restart mechanism for parallel programming systems. Master's thesis, University of Illinois, May 2000.

[12] Turing cluster. http://turing.cs.uiuc.edu/.

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