A Malleable-Job System 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

Malleable jobs are parallel programs that can change the number of processors on which they are executing at run time in response to an external command. One of the advantages of such jobs is that a job scheduler for malleable jobs can provide improved system utilization and average response time over a scheduler for traditional jobs. In this paper, we present a programming system for creating malleable jobs that is more general than other current malleable systems. In particular, it is not limited to the master-worker paradigm or the Fortran SPMD programming model, but can also support general purpose parallel programs including those written in MPI and Charm++, and has built-in migration and load-balancing, among other features.

1. Introduction

Current multiprogrammed parallel systems aim at maximizing throughput and minimizing idle time. Under space-sharing scheduling policies incoming jobs are assigned to a subset of processors. The rigid and inflexible nature of the jobs can greatly reduce the throughput of the parallel system. For example a parallel system with 100 processors running a single 64 processor job must make a new job that requires 40 processors wait for the completion of the first job. This wastage can be avoided if other smaller jobs are ready to be executed [18]. But this may not always be the case, leading to under-utilization of the system. Malleable jobs provide an excellent solution to this problem.

A useful classification of parallel jobs from the viewpoint of scheduling is provided in [6]. The paper divides parallel jobs into four classes (i) *Rigid*, (ii) *Moldable* (iii) *Evolving*, and (iv) *Malleable*. *Rigid* jobs require a fixed number of processors and cannot execute on fewer or more processors. *Moldable* jobs are flexible in the number of processors at the time the job starts, but cannot be reconfigured during execution. Both *Evolving* and *Malleable* jobs can change their processor requirements during execution. For evolving jobs [8, 7, 23] changes are application initiated. If the system cannot satisfy the job's demand, the job cannot proceed. For malleable jobs, the decision to change the number of processors is made by an external job scheduler. The distinction between evolving and malleable jobs is fuzzy. For example, in [8] the evolving job framework supports special performance monitoring infrastructure which enables programs to decide when they should change the number of processors they are running on. Although there is no external scheduler, jobs react to the system load. Therefore, we use the term *Adaptive Job* to denote jobs that are malleable and/or evolving.

This paper presents a framework that supports adaptive jobs. Both traditional adaptive MPI programs and adaptive programs written in Charm++ [11, 14], a parallel object language, have been implemented. This paper also presents an Adaptive Job Scheduler which manages these adaptive jobs so as to maximize the utilization of the system and also improve the response time of the jobs. In the above example, our system would enable job B to be started after job A has been shrunk to 60 processors, increasing utilization of the system.

Figure 1 shows the system components: (1) the *Scheduler*, (2) the parallel job runtime support (RTS) which enables adaptive migration, and (3) the job-submission *client* that remotely submits jobs and monitors them. The scheduler (one per workstation cluster or parallel machine) runs as a server listening on a well-known port. A client connects to it and requests execution of a job. After some negotiation, the scheduler might accept the job, which it starts on the cluster and manages via a network connection. Processor assignment is communicated by the scheduler to the RTS component of each job.

With the trend towards supercomputers becoming centers that disperse compute power for profit, adaptive jobs and schedulers could play an important role.

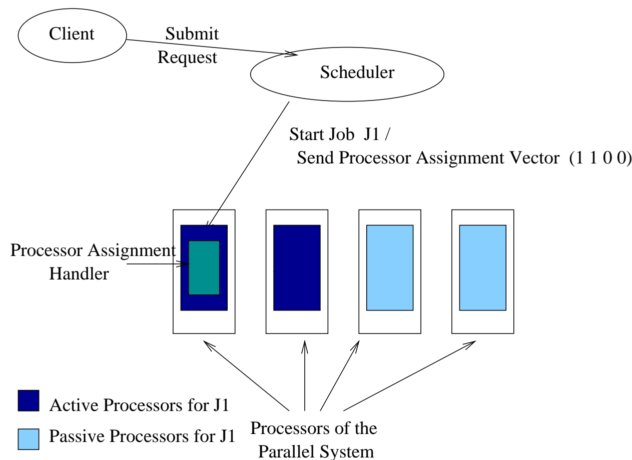

Figure 1. System Components.

In this scenario, users 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 *Computational Grid* will run the job on some available parallel computer, upload files to it, and download results back to the desktop, when necessary. The parallel computers themselves will be run as *for-profit* centers which will charge for the compute power used by applications in some fashion.[1]

The rest of this paper is organized as follows. The next section describes related work in developing malleable jobs. Section 3 describes the adaptive runtime system that can redistribute work (MPI threads or objects) to a given set of processors on demand, and also presents measurements of the overhead and interference of adaptive jobs. Section 4 presents an Adaptive Job Scheduler that shrinks and expands running adaptive jobs to ensure high utilization. The section also presents performance results of our scheduler that show an improved average response time and system utilization. We then present some ideas for future work and conclude in Section 5.

2. Related Work

Malleable and evolving jobs have been extensively studied by the scheduling and load-balancing communities. Several adaptive systems have been implemented, many of which are prototypes developed for experimental purposes. Most such systems belong to one of the following categories:

1. Master-slave programming model (also called task-queue, master-worker) in which the granularity at which reconfiguration takes place is a task or thread. [24, 23, 9]

2. Shared-memory model in which automatic compiler loop-level parallelism is exploited [8, 7]. The granularity of work distribution is a set of iterations.

3. Fortran SPMD data-partitioning style of programming, where the data is partitioned across the processors, and a reconfiguration involves the global redistribution of data. [17, 5]

We believe that these programming models are unnecessarily restrictive, especially for dynamic, irregular applications, and force the programmer to use a specific programming model.

Some other related systems are Dome [19] and Condor [16]. Dome [19] is an object oriented distributed framework where applications are load-balanced by migrating parts (i.e. data members) of objects between processors in order to improve the overall execution time. A class-specific load balancer has to be implemented for each class. Condor [16] supports runtime migration of a process from one workstation to another through checkpointing. This system only migrates sequential programs executing on one processor. CARMI [23], referenced above in the master-slave model, is based on Condor.

Our adaptive system, which is based on the Charm++ runtime system, allows the user to select their desired programming model, even to the extent of implementing different parts of their program in different models. We provide the MPI model (without requiring the master-slave style), and we also provide the object-oriented, message-driven model of the Charm++ programming language. The system is extendable and more models can be added. In both models, MPI and Charm++, the granularity of a *task* can be very small. The system handles the mapping of work to processors, measurement-based load-balancing and migration of objects/threads. The runtime system also responds to scheduler requests for dynamic re-sizing in a transparent manner.

Much work has been done on scheduling adaptive jobs. In general schedulers for adaptive jobs perform better than those for rigid jobs [24, 15, 21, 10, 20, 8, 7]. Various criteria have been used to compare scheduling algorithms, including system utilization, mean response time, meeting real-time deadlines, etc. (As an aside, it is

[1] One can then imagine that funding agencies 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.

interesting to note that backfilling [18] becomes less relevant for adaptive jobs, since the existing jobs can either expand to occupy holes, or permit a new job to occupy the hole and expand later.) Although many interesting scheduling algorithms are presented, the performance of each seems to depend on the workload (size of programs in time and space), system load, and reconfiguration overheads [9, 3]. We conclude that dynamic equipartitioning [21, 3] is a reasonably good algorithm, and we currently use a variant of equipartitioning to schedule adaptive jobs for our system.

3. Adaptive Jobs

An adaptive job is a parallel program that can dynamically (i.e. at run-time) shrink or expand the number of processors it is running on, in response to an external command or an internal event. The number of processors can vary within the bounds specified when the job is started. Typically, the user will specify the bounds taking into consideration memory usage and efficiency of the job on a given number of processors.

3.1 MPI Adaptive Jobs

Traditional MPI jobs, using a conventional implementation of MPI, are incapable of adaptive behavior. We use an adaptive implementation of MPI (AMPI [1]) to dynamically change the set of processors being actively used by a job. To use AMPI, a Fortran 90 MPI program does not need to be changed at all. It is preprocessed by a source to source translator and linked with the AMPI library, instead of the usual MPI library. C based MPI programs have to be modified, but the modification is simple and mechanical: All global variables must be encapsulated in a dynamically allocated structure. This is necessary because AMPI allows multiple virtual processes per processor, and each process must have its global variables separate from the others. For Fortran90, our preprocessor makes the necessary transformation. Similar translation can also be done for C programs, with additional compiler-preprocessor support.

AMPI [1] programs consist of a large number of *virtual* (or logical) MPI *processes*, implemented as user-level threads. The number of such threads is typically much larger than the number of processors. User programs are not aware of the physical processor on which each thread is running and communicate using the rank of the corresponding MPI virtual process only. This virtualization provides the system the ability to dynamically adapt its behavior.

In addition to AMPI, adaptive jobs may also be written using Charm++, a parallel C++ system that supports virtualization and data driven objects [12].

3.2 The Runtime System

Adaptive programs (both AMPI and Charm++) are implemented on top of the Charm++ [13] runtime system. Charm++ provides a sophisticated load balancing framework. The load balancing framework keeps track of the load presented by each thread and object, and when triggered by either an internal or external trigger, redistributes the threads and objects to balance the load. AMPI uses a sophisticated scheme to permit migration of user level threads, as described in [1]. The load balancing framework also supports plugin strategies that take relative processor performance and background load into account. So more load will be allocated to faster processors and/or processors with low background load.

We modified this framework so that it accepts a processor map and allocates work to only the processors enabled in the processor map. The current implementation uses a centralized load balancer and load balancing is done on 'processor 0' by default. If 'processor 0' is not set in the processor map the first processor in the processor map becomes the new load balancer.

When a new parallel job is created it is started on all the processors in the system but load is only allocated to the processors enabled in the processor map (if the number of processors is large, the job could be started on a partition but the expansion of the job would be restricted to that partition). The run-time system maps the threads and objects to physical processors under the control of a load balancer.

When the processor map is changed either by the application or by an external job scheduler the load balancing framework triggers a thread-migration phase to move the threads out from the vacated processors. A skeleton process is left behind on each vacated processor to forward messages meant for objects/threads that were previously housed on that processor. The overhead of this process is very small and consists of a transient period of forwarding messages, and periodic (but nominal) participation in global operations such as reductions and load balancing. The above features of the load balancing framework enable evolving programs to be easily written.

We use the Converse[13] client-server interface, which allows an external client to inject a message into a running parallel application over the network, to inform the load balancer module when the processor map changes. The new processor map is sent to the load bal-

ancing framework by the Adaptive Job Scheduler. As in the case of evolving jobs the load balancer triggers a thread-migration phase to move the threads out from the vacated processors and leaves a skeleton process behind on the vacated processors.

Two questions naturally arise about the overhead of this method: (1) how quickly can we shrink (or expand) a job? and (2) how much interference do the residual processes cause to the performance of another job on the same processor? We now present experimental results to quantitatively answer these questions.

3.3. Performance

To test our Adaptive Job system, we conducted several experiments on the NCSA Platinum Cluster (a Linux cluster with 512 dual-processor 1 Ghz Pentium III nodes connected by Myrinet) and the PSC TCS cluster (which consists of 750 quad-processor AlphaServer systems running Tru64 UNIX and connected by elan). The benchmark adaptive job we used was a molecular dynamics (MD) program (a simplified version of NAMD[2]). We found that two factors affected the shrink/expand time of the adaptive jobs. The first factor is the *migrated data size*, which is the *amount of memory per processor* that needs to be migrated when the job shrinks or expands and the second factor is the number of processors allocated to the job.

So we performed two sets of experiments in which we varied one of the above mentioned two parameters while keeping the other fixed. In the first set of experiments the number of processors is varied for two migrated data sizes of 1MB and 10MB respectively (total memory sizes being 6MB and 60 MB per processor, the additional memory in excess of migrated data size was used by the MD program for messaging and buffering). Each row in the Tables 1, 2, 3, 4 presents both the shrink and expand times; e.g. it takes 86 ms to shrink a job from 128 to 64 processors on the platinum cluster and 67.6 ms to expand it back to 128 processors. The second set of experiments vary the migrated data size for 16 and 64 processors; Figure 2 shows the results.

As seen in Tables 1, 2, 3, 4 and Figure 2, the adaptation time is small and easily scales to 128 processors with a total migrated data size of 1.28 GB (Tables 2, 4), thus making it feasible to shrink and expand jobs at each scheduling decision (which typically occurs several minutes apart).[2]

When shrunk, the job leaves a residual process on the processors it vacates as described in Section 3.2. Fig-

[2]We are not sure about the reasons for the irregular pattern in shrink/expand times. We think it might have to do with network congestion, since we did not run the system in dedicated mode.

Processors	Shrink Time (ms)	Expand Time (ms)
128 to 64	86.0	67.6
64 to 32	74.3	57.3
32 to 16	72.2	50.4
16 to 8	75.0	54.2
8 to 4	65.7	49.7

Table 1. MD Program with a 1MB migrated data size on NCSA Platinum

Processors	Shrink Time (ms)	Expand Time (ms)
128 to 64	614	502
64 to 32	660	538
32 to 16	696	506
16 to 8	594	461
8 to 4	564	489

Table 2. MD Program with a 10MB migrated data size on NCSA Platinum

Processors	Shrink Time (ms)	Expand Time (ms)
256 to 128	253.3	84.4
128 to 64	165.1	90
64 to 32	256.4	182.1
32 to 16	375.4	261.7
16 to 8	185.4	118.7

Table 3. MD Program with a 1MB migrated data size on PSC TCS

Processors	Shrink Time (ms)	Expand Time (ms)
128 to 64	2116	790.2
64 to 32	766.8	874
32 to 16	637	516.2
16 to 8	559.6	382

Table 4. MD Program with a 10MB migrated data size on PSC TCS

#Jobs in the system	Performance Cost
2	1.98 %
4	1.43 %
8	3.24 %

Table 5. Adaptive Job Performance Cost

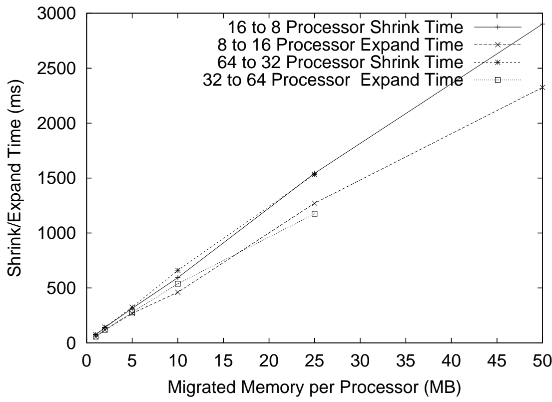

Figure 2. Shrink and Expand Times on the Platinum Cluster at NCSA

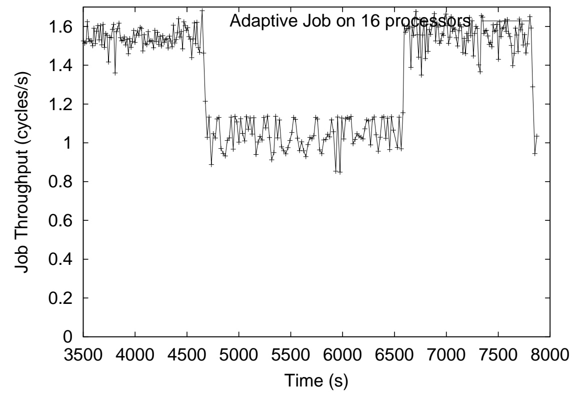

Figure 5. Job Throughput for an Adaptive Job

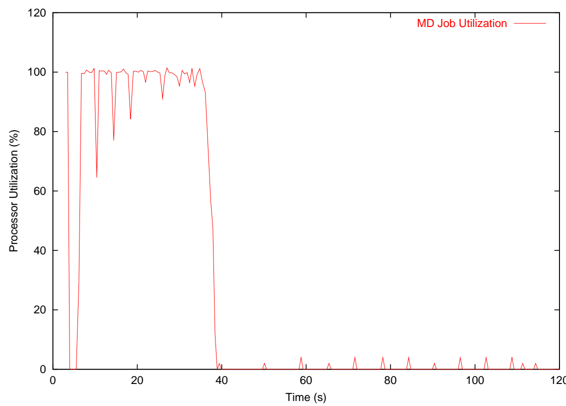

Figure 3. Residual Load after Shrinking.

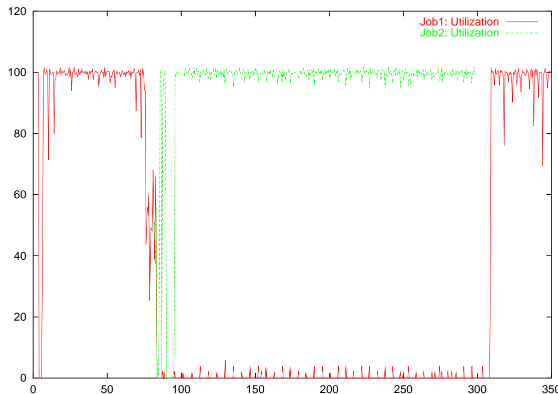

Figure 4. Proof of Concept of Shrink/Expand.

ure 3 shows the load on one of the processors after the job has been vacated at time 38. This load is zero most of the time but has periodic peaks of about 2%. Figure 5 shows the throughput of another adaptive job running on 16 processors after it has been shrunk to 8 processors and then expanded to 16 processors. Figure 4 shows the utilization of a processor by two adaptive jobs. When Job2 arrives, the Adaptive Job Scheduler first shrinks Job1 and then starts Job2. In this way the shrinking of Job1 is overlapped with the start up time of Job2 and load sharing between the jobs is minimized. When Job2 finishes, the scheduler asks Job1 to expand. Thus, despite the presence of Job1's residual process, Job2 gets most of the CPU. Apart from occupying virtual memory, the impact of residual processes is very minor. We plan to do some work on eliminating them.

Table 5 shows the performance drop due to the interference by other adaptive jobs. The table presents the performance cost of running two *8-processor*, four *4-processor* and eight *2-processor* adaptive jobs on a 16 processor cluster, as a percent increase in total execution time, e.g. running two 8-processor adaptive jobs together takes 1.98 percent more time than running two 8-processor rigid jobs. As can be seen in Table 5 the loss of performance is very small even for a reasonable number of jobs present in the system.

4. The Adaptive Job Scheduler

To test our Adaptive Job System we developed an Adaptive Job Scheduler. In this paper we present results from our Adaptive Job Scheduler with a simple scheduling strategy which is a variant of equipartitioning [21, 3, 24, 9]. Each incoming job specifies the min-

imum and maximum number of processors it can use. When a new job arrives [3], the scheduler re-calculates the number of processors allocated to each running job. All jobs, including the new one, are allocated their minimum number of processors. Leftover processors are shared equally, subject to each job's maximum processor usage. If it is not possible to allocate the new job its minimum number of processors, it is enqueued. When a running job finishes, the scheduler applies the above algorithm to each job in the queue again.

After running this algorithm some jobs might shrink, some might expand, and some might remain unchanged. The results are communicated to the running jobs by sending each a bit-vector of the processors available to the job. The jobs will then resize themselves.

The study in [3], presents the performance gains of the dynamic equipartitioning strategy, with an overhead of a 5 second stall for each reconfiguration. Based on the performance data presented in 3.3 our system comfortably meets the 5 second requirement for most jobs. We will also show in the next section that the simple scheduling strategy mentioned above also provides improved performance over traditional First Fit queuing systems (such as DQS [4], PBS [22], etc.) Our scheduler infrastructure allows us to plug in different scheduling strategies, and more sophisticated strategies are being explored.

4.1. Adaptive Job Scheduler Performance Results

To demonstrate the effectiveness of our Adaptive Job System we performed experiments on a parallel Linux cluster [25]. These experiments were performed separately for both adaptive and traditional (rigid) jobs. The benchmark program used for the experiments was the same MD program described in Section 3.3, but with a smaller problem size of 50,000 atoms (about 5MB total memory). The program takes approximately 64.5 seconds to complete 100 iterations on 64 processors. It uses a naive parallel algorithm, and has a sub-linear speedup characteristic, as shown in Figure 6. We used it as a realistic example application.

4.1.1 Experiments on the Linux Cluster

The experiments were performed on a Linux cluster with 32 dual 1 GHz Pentium III nodes connected by 100 Mbps Ethernet. A random job generator was used to fire jobs to the scheduler and job arrival was Poisson distributed. Each job submission ran the benchmark program mentioned above for different number

[3]On a production system, the frequency of job arrivals is one every several minutes.

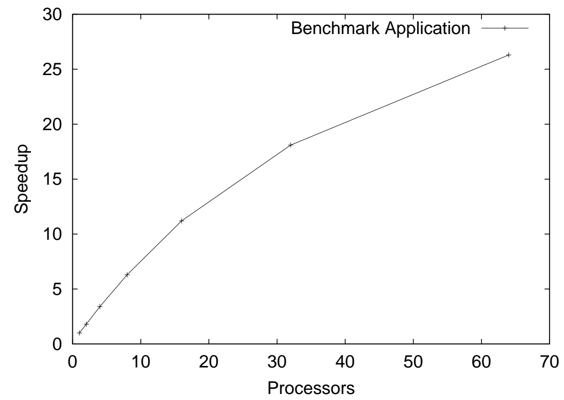

Figure 6. Benchmark Program Speedup

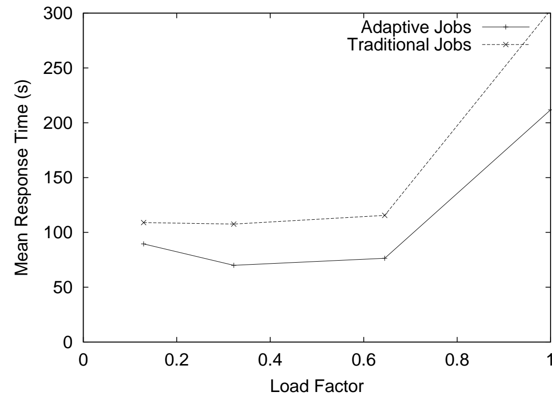

Figure 7. Mean Response Time on the Linux cluster with benchmark application

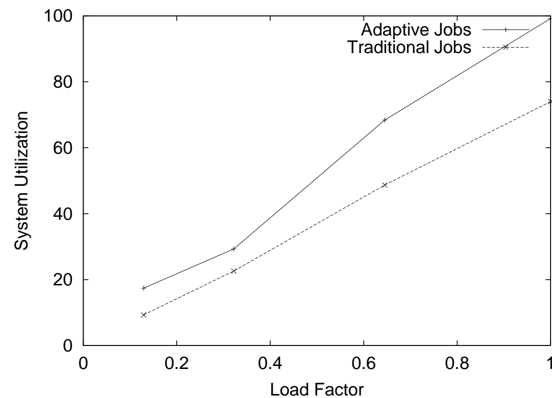

Figure 8. System Utilization on the Linux cluster with benchmark application

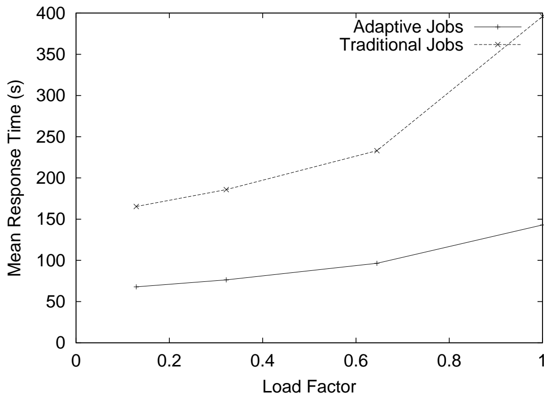

Figure 9. Mean Response Time (MRT) for simulated jobs with sub-linear Speedup

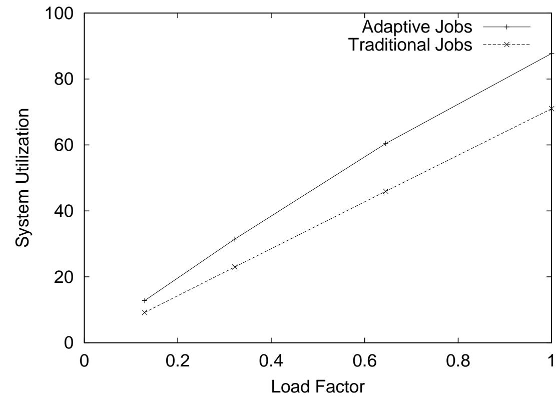

Figure 10. System Utilization for simulated jobs with sub-linear Speedup

of iterations. The number of iterations with an exponentially distributed with a mean of 100 iterations, to model an exponential service time. The experiments computed the mean *Response Time* and the mean *System Utilization*. All experiments had 50 job arrivals and the results are presented in Figures 7 and 8. Here lf is the mean load factor (i.e. $lf = \lambda * (execution\ time\ on\ 64\ processors)$, where λ is the mean arrival rate). The traditional jobs were submitted to a scheduler that emulates traditional schedulers like PBS [22] and DQS [4]. Adaptive jobs were submitted to our Adaptive Job Scheduler.

In these experiments the *minpe* (minimum number of processors requested by each job) of the adaptive jobs is uniformly distributed from 1 to 64 and the *maxpe* is set to 64. For the traditional jobs the the number of processors is uniformly distributed between 1 and 64.

4.1.2 Simulations

We believe that the performance of the Adaptive Job System would improve with the number of jobs submitted. So we performed simulations to compute the mean *Response Time* and the mean *System Utilization* after 10,000 job submissions. The simulations modeled a cluster with 64 processors with an Adaptive Job Scheduler and a Traditional Job Scheduler. The simulations used the same random number generator for generating job arrival times and execution times as the Linux cluster experiments. The simulations also modeled the same benchmark program mentioned above. The results of the simulations are shown in Figures 9 and 10 which present the mean response time and the mean utilization of the system respectively, for different load factors. The simulations reflect the long-term steady state performance gains our Adaptive Job System.

5. Summary and Future Work

We described the motivation, design, and implementation of an *adaptive* job system, and an Adaptive Job Scheduler for parallel machines. Adaptive applications can be written in a variety of languages including MPI and Charm++. The adaptive system builds upon a measurement based load balancing system. The original load balancers, which aimed at resolving application induced load imbalances, were extended to shrink, expand or to change the set of processors allocated to a job. This is accomplished by migrating user level entities (such as MPI threads and Charm++ objects) across processors. Mechanisms for controlling the behavior of the load balancer via bit vectors of available processors were implemented and validated. We also described and implemented a simple job scheduling strategy, and presented some 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 system 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 the IBM SP, which allows socket based communication with outside processes. We plan to utilize Globus components, and make the job scheduler a Globus server. We also intend to develop techniques to eliminate the residual processes left behind when an

adaptive job vacates a processor, in order to reduce the admittedly tiny residual load. We expect our scheduler to be in production use on the 400 processor CSE cluster at Illinois for running ASCI Center and CSE jobs.

The software can be downloaded at our website: http://charm.cs.uiuc.edu/. and a demonstration of the *Faucets* project is also available by clicking on the Faucets link.

References

[1] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[2] J. A. Board, L. V. Kalé, K. Schulten, R. Skeel, and T. Schlick. Modeling biomolecules: Larger scales, longer durations. *IEEE Computational Science and Engineering*, 1(4), 1994.

[3] S.-H. Chiang and M. K. Vernon. Dynamic vs. static quantum-based parallel processor allocation. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 200–223. Springer-Verlag, 1996.

[4] D. W. Duke, T. P. Green, and J. L. Pasko. Research toward a heterogenous networked computing cluster: The distributed queueing system version 3.0. Technical report, Florida State University, May 1994.

[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] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1996.

[7] M. W. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. *Concurrency: Practice and Experience*, 10(14):1235–1250, 1998.

[8] S. Ioannidis, U. Rencuzogullari, R. Stets, and S. Dwarkadas. CRAUL: Compiler and run-time integration for adaptation under load. *Journal of Scientific Programming*, Aug. 1999. Invited paper.

[9] N. Islam, A. Prodromidis, and M. Squillante. Dynamic partitioning in different distributed memory environments. In *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[10] Jansen and Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[11] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

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

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

[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] W. Y. Lee, S. J. Hong, and J. Kim. On-line scheduling of scalable real-time tasks on multiprocessor systems.

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

[17] 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.

[18] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. on Parallel and Distributed Systems*, 12(6), June 2001.

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

[20] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Using runtime measured workload characteristics in parallel processor scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[21] J. Padhye and L. Dowdy. Preemptive versus non-preemptive processor allocation policies for message passing parallel computers: An empirical comparison. In *Proceedings of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, Apr. 1996.

[22] Portable batch system. http://pbs.mrj.com/.

[23] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with CARMI. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 259–278. Springer, 1995.

[24] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM SIGOPS Symposium on Operating Systems Principles*, Dec. 1989.

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