Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

SIMULATION OF JOB SCHEDULING FOR SMALL SCALE CLUSTERS

Hassan Rajaei
Mohammad Dadfar
Pankaj Joshi

Dept. of Computer Science
Bowling Green State University
Bowling Green, OH 43403, U.S.A.

ABSTRACT

Despite growing popularity of small-scale clusters built out of off-the-shelf components, there has been little research on how these small-scale clusters behave under different scheduling policies. Batch scheduling policies with backfilling provide excellent space-sharing strategy for parallel jobs. However, as the performances of uniprocessor and symmetric multiprocessor have improved with time-sharing scheduling strategies, it is intuitive that the performance of a cluster of PCs with distributed memory may also improve with time-sharing strategies, or a combination of time-sharing and space-sharing strategies. Apart from the batch scheduling policies, this research explores the possibilities of using synchronized time-sharing scheduling algorithms for clusters. This paper describes simulation of the Gang scheduling policies on top of an existing batch scheme. The simulation results indicate that time-sharing scheduler for clusters could exhibit superior performance over a batch policy.

1 INTRODUCTION

With the increase in processing power and decrease in the prices of today's commercial-off-the-shelf (COTS) PCs, and the increase in the bandwidth of easily available and affordable Ethernet, "home grown" clusters like Beowulf cluster have been popularly built for High-Performance-Computing and Parallel Computing environment. Such parallel processing finds a wide range of usage among researchers and practitioners. However, often the parallel systems suffer from under utilization due to inappropriate choice of scheduling policy. A scheduling policy is used to settle the conflicts in resource acquisition when a job requires more nodes than that are currently available.

In our previous studies (Rajaei and Dadfar 2005, 2006), various backfilling techniques, namely conservative and aggressive backfilling (Srinivasan et al. 2002), multiple-queue (Lawson and Smirni 2002) and lookahead (Shmueli and Feitelson 2003), have been investigated. Multiple queue backfilling suffers from over-fragmentation of available nodes in a small-scale cluster (Rajaei and Dadfar 2006), whereas other techniques seem to be more promising. The main drawback of backfilling is higher response time for some jobs which are requesting more resources particularly in case of aggressive backfilling, and some jobs may suffer from overestimation of required processing time.

Gang scheduling (Zang et al. 2003) overcomes the problem of response time, while its disadvantage is the global synchronization overhead needed to coordinate a set of processes. Even though it incurs a heavy context switch overhead, it may still serve as a viable scheduling alternative in small-scale clusters. Gang scheduling also offers various tunable parameters, which can be dynamically changed according to size and nature of workload. The number of time slots, which are the interval allocated to a job during which it runs without preemption, and their duration can be fine-tuned dynamically. Gang scheduling also supports dynamic prioritization of jobs.

This research simulates Gang scheduling based upon various policies and tries to find its adaptability to small-scale cluster. Arrival of the jobs to the system as well as the payloads and other interesting attributes are randomly generated. The generated jobs are sent to the simulated scheduler who in turn activates the desired nodes based on its current policy.

The goal of this research is to provide a framework that can be used beyond the current simulation of scheduling policies. We investigate whether a timesharing policy is suitable also for cluster computing. If the answer is positive, then we could replace the batch scheduling with an appropriate scheme satisfying the selection criteria. Other parameters such as migration policies and cost performance analysis constitute important research elements.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 provides details of the scheduling polices under investigation. The simulation

methods are discussed in Section 4 and details of the implementation is provided in Section 5. Results and analysis are discussed in Section 6. Extension to the current work is described in Section 7 and finally, concluding remarks are provided in Section 8.

2 RELATED WORK

Various types of scheduling policies have been implemented, especially for large-scale clusters - ranging from policies based on space-sharing to time-sharing to a combination of both. Schedulers such as EASY (Srinivasan et al. 2002), do not support prioritization. Talby and Feitelson (1999) advocate use of priorities for more effective backfilling policy. When a new job is submitted, all possible schedules are priced according to utilization and priority considerations and as long as no job is delayed beyond its *slack* which is a time duration that determines how long a job may have to wait before running. The *slack* is comparable to the term *shadow time* (discussed later in Section 3.1) but with additional consideration for priorities. One problem associated with slack-based policies is the consideration of all possible scheduling which appears to be an NP-hard problem, and hence very time consuming.

Multiple-queue backfilling (Lawson and Smirni 2002) is based on aggressive backfilling and aims at reducing fragmentation of system resources by dividing the system into multiple disjoint partitions by job category that depends on estimated job duration. However, in small clusters our research suggests that the multiple-queue policy may increase the system fragmentation contrast to what the policy is advocating (Rajaei and Dadfar 2006). Nevertheless, we anticipate that the algorithm might show better performance in large clusters or grids.

Attempts have been made to integrate gang scheduling, backfilling and migration (Zhang et al. 2003) in order to alleviate the problem with space sharing. For small clusters, integrating gang scheduling with backfilling holds promise mostly because of low overhead in the global context switch. Current research explores this option, while migration choice is left for future work.

With gang scheduling, processes involved in I/O or blocking communication can make processors idle. Gang scheduling policy in pairs (Uwe and Ramin 1998) aims at alleviating this problem. The scheme pairs the gangs that make heavy use of the CPU with the gangs that are intensively using I/O or other blocking operations.

3 SCHEDULING POLICIES

In this section we consider three scheduling policies chosen for our simulation studies: one batch scheduling with backfilling and two gang scheduling policies; greedy and backfilled.

3.1 Batch Scheduling: Backfilling Lookahead

Batch scheduling is non-preemptive and once a job is started, it runs until it completes its execution. Simple batch scheduling algorithms like First Come First Serve (FCFS) and Shortest Job Next (SJN) may waste processing time when the first job cannot run in FCFS or longer jobs may suffer from starvation using SJN. To overcome such problems, various types of backfilling algorithms have been used to allow small jobs from the back of the queue to bypass the long jobs which are waiting for resource availability.

Unlike other backfilling policies, i.e. aggressive and conservative, which consider the queued jobs one at a time, backfilling with lookahead bases its scheduling decisions on the whole contents of the queue. The waiting queue is processed using a dynamic-programming based scheduling algorithm that chooses the set of jobs which will maximize the machine utilization and will not violate the reservation for the first waiting job.

For this research, we have simulated what Shmueli and Feitelson (2003) call the basic algorithm. It tries to find a combination of jobs that together maximize utilization without violating the prior reservation made by the job which is at the head of the wait queue. It uses dynamic programming approach for the aggressive backfilling. Our previous research (Rajaei and Dadfar 2005) suggested that there is very little to choose between conservative and aggressive backfilling, but the complexity of efficient implementation of conservative backfilling is far greater. So we consider the basic algorithm, which is based on aggressive backfilling, to represent the basic characteristics of all backfilling policies.

The backfilling lookahead algorithm executes the job at the head of the wait queue if enough nodes are available. Otherwise it calculates the shadow time, which is the earliest time at which enough free nodes will be available for the job at the head of the wait queue to get executed. If any other job in the wait queue can execute with the currently available nodes and get completed before the shadow time, then it is executed.

3.2 Gang Scheduling

Gang scheduling refers to a policy where all processes of a parallel application are grouped into a gang and simultaneously scheduled on distinct processors of a parallel computer system such as a Beowulf cluster. Multiple gangs may execute concurrently by space-sharing the resources. Furthermore, division of the system according to time slots is supported through synchronized preemption and later rescheduling of the gang. Context switching is coordinated across the nodes such that all the processes are scheduled and de-scheduled at the same time. At the end of a time slot, the running gangs get blocked allowing other gangs to

run. One important promise of the gang scheduling regards better rescore utilization for parallel programs across the available compute nodes.

There are three synchronization options:

1. Achieved through synchronized clocks. (SHARE Scheduler IBM SP2). The nodes do not interact through explicit synchronization, and do not receive any coordination message from the central scheduler.
2. Coordinated by the master node. (ParPar Scheduler or Score-D). We choose this option which suits better for our cluster.
3. SCORE-D uses a high performance communication library called PM which supports network context switching (Choi et al. 2004).

The number of time slots *n* is limited to a number supplied by the user. This number should be kept moderate since increasing it would result in a job having to wait longer for its turn to run which can be unacceptable. The maximum time a job will have to wait after it is being preempted, to get rescheduled, will be $(n-1)*tq$, where tq is the time quantum of each time slot. The maximum time can be reduced by reducing tq, but it will result in increased number of context switches which is unacceptable.

3.2.1 Gang Scheduling with Greedy Approach

With the greedy approach, all jobs in the waiting queue are considered as suitable candidates for execution. The jobs that have the required number of nodes in any time slots are executed. The policy does not take into consideration of the arrival time of the jobs nor does it consider the estimated end-time. As with any greedy approach, the resulting schedule may not be fair.

3.2.2 Gang Scheduling with Backfilling

The jobs in the waiting queue are considered as per the lookahead backfilling policy. The job at head of the waiting queue has the reservation of the nodes it requires, and that reservation is not violated by the jobs that arrive later even if they get executed earlier than the first waiting job. This approach is fairer than the greedy approach, but not as fair as the conservative backfilling approach.

The states in which the processes of a job can be at the various worker nodes are shown in Figure 1. Job1 runs in the time slot 0; i.e. all the processes belonging to Job 1 are in running state at time slot 0. With the arrival of context switch event from the scheduler, Job1 is blocked and Job3 is unblocked. An unblocked process enters the ready state. But assuming no other active process is running in the worker node, the unblocked process readily enters the 'running' state or starts running.

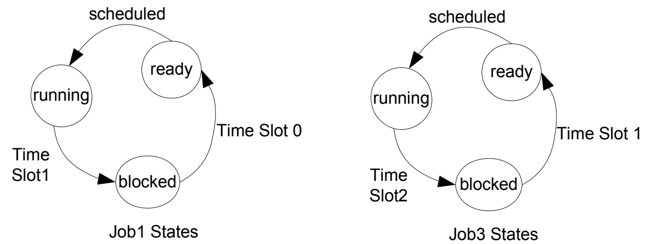

Figure 1: States Changes and Time-Slots in Gang Scheduling with Backfilling for Two Jobs

4 SIMULATION METHODS

The architecture of the simulator is shown in Figure 2. A simulator for the scheduler is created on top of the Message Passing Interface (MPI) for various message passing and synchronization purposes of the simulated scheduler. The simulation program itself is a parallel job to the cluster. It consists of one dedicated scheduler process and several application processes. The Portable Batch Scheduler (PBS) (Bode et al. 1999) is used to launch the simulator from the server to the compute nodes of the cluster. The PBS script reserves all the nodes and dispatches the job.

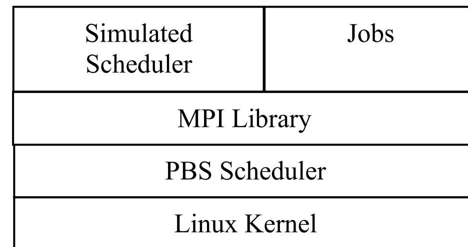

Figure 2: Simulator Architecture

Since our simulator runs on top of MPI/PBS, we use one node as the simulated scheduler and the rest as the simulated computed nodes. All 16 nodes of the cluster are reserved using the PBS commands. As an MPI application, the program lets Node 0 to act as a scheduler, while all the other 15 nodes wait for messages from the scheduler. The simulator accepts jobs from the user (Figure 3). Depending on the simulated policy, the scheduler allocates the required number of nodes for the job from the available resources among the 15 workers. For the gang scheduling policy, the scheduler also allocates the time slot for the job.

The general modes of operation for evaluating the scheduling policies are:

1. Generate workloads.
2. Simulate the behavior of various scheduling policies while running the simulated workloads.
3. Determine various parameters of interest for each scheduling policy.

Characteristics of the simulated jobs in this study are considered as follows:

- Independent: Each job is independent of the other. That is, the order in which the jobs are executed should not impact execution of other jobs.
- Rigid: Each job has a fixed number of processes that does not change during execution.
- Deterministic: Each job will specify the estimated time for which it wants to remain in they system.

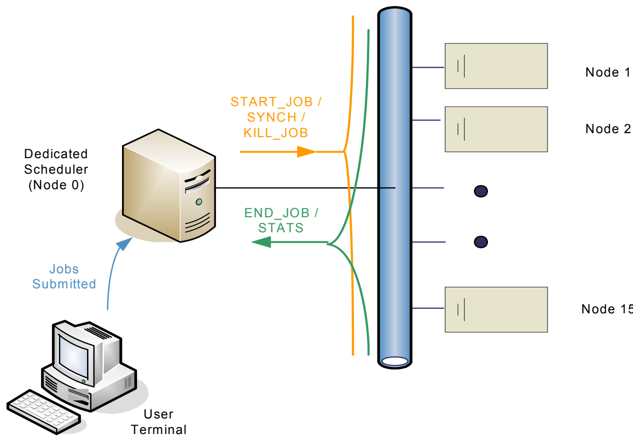

Figure 3: Overview of the Simulated Scheduler

For the gang scheduler, at the end of each time quantum, the simulated scheduler broadcasts a SWITCH_CONTEXT message using scatter provided by the communicator class. The message contains information about which time slot is to be scheduled next (e.g. slot # 3 in Figure 4). On receiving the context switch command from the scheduler, each node stops the currently running process using SIGSTOP, and resumes the jobs scheduled to run next using the signal SIGCONT**.**

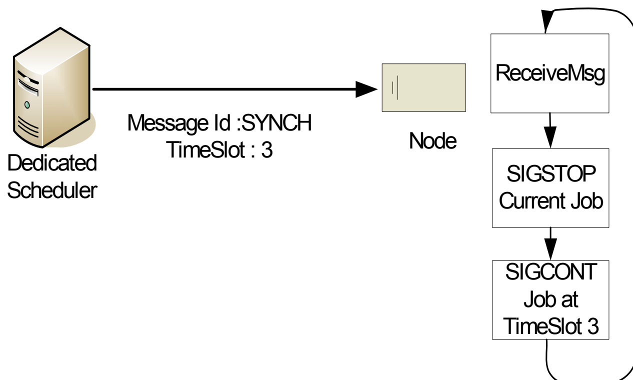

Figure 4: Scheduler Signals for Context Switch

Experiments were carried out with a randomly generated workload. The arrival time, estimated execution time of the jobs (submitted by the user), the actual simulated run-time by the simulator, and the number of requested nodes were generated randomly.

5 IMPLEMENTATION

In this study, three scheduling policies were simulated, namely Lookahead Backfilling Policy, Gang Scheduling with Greedy Approach and Gang Scheduling with Lookahead Backfilling Policy. This section provides some details of the implementations.

5.1 The Environment

Our cluster has the following system features:

- 16 homogeneous compute nodes.
- 2.8 GHz Pentium 4 processor per node .
- 1 GB of RAM per node.
- 1 GB Ethernet switch.
- Linux operating system with Gentoo.
- Batch System with PBS based Torque.

5.2 The Simulator

The simulator is written in C++ using MPI. The base class Simulator provides some very basic functionalities of the simulation platform. It maintains an event list where events, in the form of arrival of new jobs, are inserted in the order of their arrival time. It also maintains the waiting queue where events that cannot be scheduled immediately are queued. Derived classes override *processEventQueue()* and *processWaitQueue()* methods to process the event and waiting queues.

For time management, the simulator provides a one-shot timer functionality. Subclasses need to override a method called timerFunction() which is invoked when the one-shot timer expires. The scheduler classes, which derive from the Simulator class, make use of the timer to trigger events like global context switch, start and termination of jobs. The simulation time is forwarded at the timer expiration. The resolution of simulation time and the time interval between context switches have been kept the same in this implementation for simplicity.

Jobs are generated in a pseudo-random fashion using the Linux *rand()* function in the current implementation. Other distributions, like exponential or Poisson, can be used for the study of scheduling characteristics under various workloads.

5.3 The Scheduler

It is possible to simulate batch schedulers in a uniprocessor and compare performances of various batch scheduling policies. However, for timesharing scheduling policies it is hard to make proper estimation of context switch and

communication overhead. Further, it is non trivial to make a good heuristic assumption for either the context switch or the communication overhead. Therefore, unlike our previous works, we chose to implement a simulated scheduler that manages processes across the nodes of the cluster. The context switch overhead then is what Linux scheduler enforces and the communication cost is what the underlying MPI library and the hardware entail.

The scheduler is designed atop MPI primarily because of its ease of use and its efficiency. Without MPI, using the Linux sockets for collective communication (broadcast, multicast) at the user-level would have been very inefficient considering the synchronization overhead.

6 RESULTS AND ANALYSIS

Experiments were carried out with a randomly generated workload as mentioned in Section 5.2.

A simulated policy is evaluated by scheduling criteria which reflect user's parameters of interest. A fair and quick response time is desired. Completion time of the last job, or makespan, is frequently used in research. The makespan represents the utilization throughput. In this research, the following parameters of interest have been considered:

- Makespan: Total time to completely process all jobs from a given pool of jobs.
- Wait Time (Response Time): Length of time from when a job arrives to when it enters the running state for the first time.
- The above parameters are studied to gauge the performance of various scheduling policies as the number of jobs increase, the number of time slots change, or the nature of jobs (communication oriented or compute-intensive) change.

Based on the above criteria we gathered the needed statistics and analyzed the scheduling policies. Figure 5 illustrates that the gang scheduling outperforms the backfill with lookahead in terms of both makespan and the average response time. Plotted against increasing number of jobs, the makespan for the backfill is always more than those for the gang scheduling.

Within the gang-scheduling (GS) category, the gang scheduling with greedy approach seems superior to the gang scheduling with backfill. Interestingly, GS with backfill tends to exhibit a behavior that is a compromise between backfill and GS with greedy approach. For less number of jobs, the GS with backfill coalesces with GS with greedy approach. This is because, as the number of jobs is less, time slots are readily available for most of them and neither the greedy nor the backfill policy effectively comes into play. As the number of jobs increases, they are queued and scheduling criteria are applied to pick the job to be scheduled. The greedy GS tries to schedule as many jobs as it can without consideration for fairness or reservation for the first job in the wait queue as is done by GS with backfill. It is not surprising that for a fairer scheduling policy the makespan is relatively worse but it is still better than the backfilling used with batch processing.

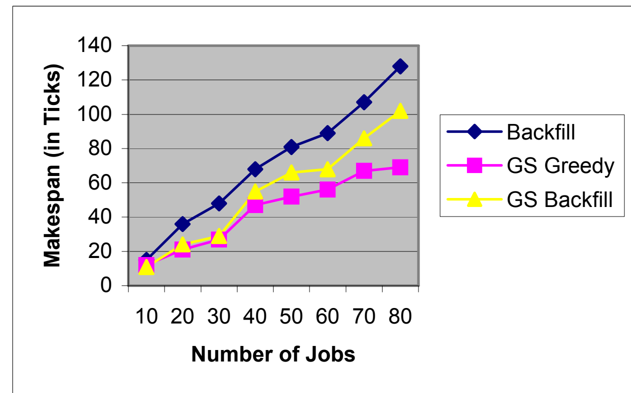

Figure 5: Makespan vs. Number of Jobs

As expected the average wait (response) time for the gang scheduling is far less than that for the backfill as shown in Figure 6. In the case of backfill, the average response time increases more rapidly making it unsuitable for interactive jobs. The gang scheduler performs appreciably, as the response time does not show a rapid increase in average response time. It also suggests that more jobs are getting completed making room for newer jobs to get scheduled.

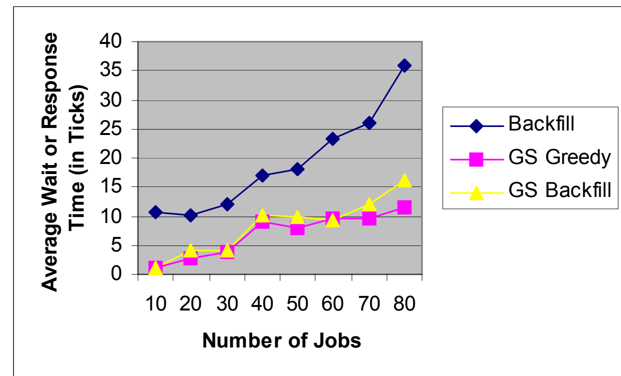

Figure 6: Average Wait Time vs. Number of Jobs

Figure 7 shows how the response time varies between a backfill scheduler and a gang scheduler. Both of the schedulers take a sample of simulated workload consisting of 80 jobs. We can see that in case of backfill, as the jobs keep arriving, the jobs that are arriving later suffer from the increase in response time. For the gang scheduler, the response time does not show such wide variation. The response time is always and consistently lower than compared to backfill. It is expected that with more jobs coming

and running for a longer time, the response times for jobs arriving late are going to increase, but the gang scheduler will consistently outperform the backfilling scheduler.

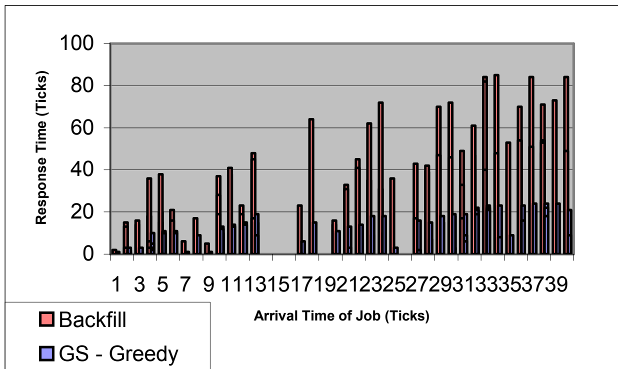

Figure 7: Response Time vs. Jobs Sorted by Arrival Time

As illustrated by Figure 8, performance of the gang scheduling exhibits appreciable improvement when the number of time slots is increased from 1 to 5. With only one time slot, the gang scheduler behaves like a batch scheduler. If the number of time slots is increased beyond 5, there is a tendency for the makespan to worsen. This can be attributed to increase in number of context switches as the number of slots increases. Overall, there is a change of about 10 ticks between when the number of slots is 5 to when it is 10 or beyond. The reason why the makespan does not deteriorate further is because even if the number of slots is increased, the number of jobs is the same, which means that some of the slots are not used at all. So even if we assign 15 slots, there might be only 10 active slots.

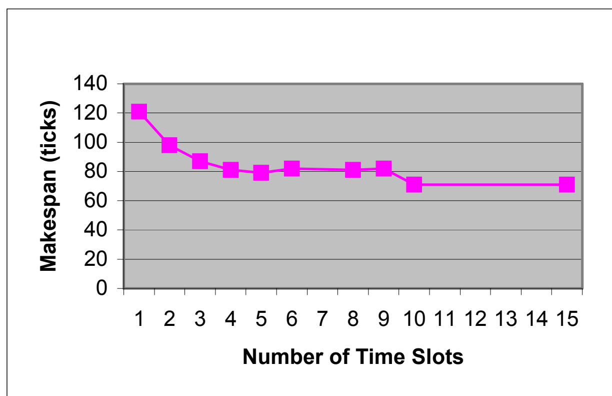

Figure 8: Makespan vs. number of time slots

As the number of time slots increases (see Figure 9), the average wait time decreases significantly between the number of slots 1 and 5. As the system behaves like having virtual nodes equal to the number of slots times the number of actual nodes, more jobs can run without any delay, thus reducing the total and average wait time.

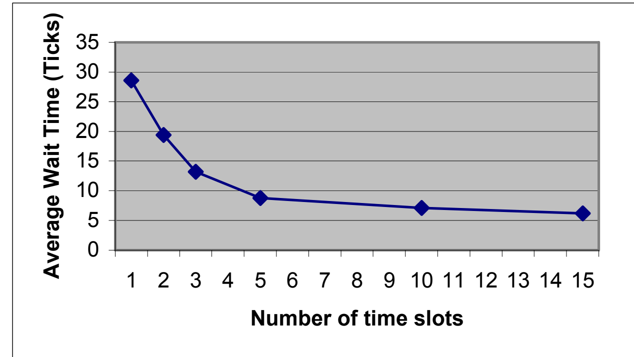

Figure 9: Average Wait Time vs. Number of Time Slots

7 FUTURE WORK

There are several interesting extensions to the current work which we plan to explore. One is the use of process migration, another one is implicit scheduling, and a third on the use of statistics based on real workload. The scalability of the scheduling algorithms needs closely to be looked at.

It has been suggested that the gang scheduling policy can be improved through the addition of migration capabilities (Zhang e al. 2003). The process of migration embodies moving a job in the Ousterhout matrix to a row in which there are enough free processors to execute that job. This will allow the row from which the job got migrated to have more free nodes and can therefore be able to run jobs which are requesting large number of nodes.

An implicit scheduling method takes a totally different approach and does not use global synchronization. Further study needs to ascertain if the implicit scheduling can be a viable option, or its complexity outweighs the required performance value.

The workload was randomly generated in this study. A real application could exhibit differently and hence impact the outcomes. We need to look at this situation as well.

8 CONCLUDING REMARKS

Gang scheduling offers an attractive solution to the drawbacks of batch scheduling, especially with respect to the response time and overestimation of the processing time of the jobs. It further introduces various tunable parameters like number of slots, the duration of slots, and job priority which can be dynamically altered to maximize the objective functions like CPU utilization. This simulation study indicates that, concurrent context switch of processes does not seem to degrade the performance, as the *makespan* did not increase alarmingly as the number of arriving jobs was increased. However, the effect of applications which require swapping when the context switch occurs needs to be investigated further. In future studies, we could build more credibility by running real workloads to the simulated scheduler or have statistics gathered from diverse real ap-

plications mapped to the arrival jobs. Consequently, we might have better proof that the performance of gang scheduling is superior to that of batch scheduling and its various flavors.

REFERENCES

Bode, B., D. M. Halstead, R. Kendall, and Z. Lei. 1999. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. In *Annual Technical Conference, USENIX 1999*.

Choi, G. S., J. Kim, D. Ersoz, A. B. Yoo, and C. R. Das. 2004. Coscheduling in Clusters: Is It a Viable Alternative?, In *Proceedings of Super Computing (SC).*

Frachtenberg E., F. Petrini, S. Coll, and W. C. Feng. 2001. Gang Scheduling with Lightweight User-Level Communication. *International Conference on Parallel Processing (ICPP) Workshops*, pp. 339-348.

Góes, L. F. W., and C. A. P. S. Martins. 2004. Reconfigurable Gang Scheduling Algorithm. *10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. LNCS.

Gonzalez Jr., Mario. 1997. Deterministic Processor Scheduling. *ACM Computing Surveys, Vol.9, No.3*.

Hori, A., H. Tezuka, and Y. Ishikawa. 1998. Highly Efficient Gang Scheduling Implementation. In *Proceedings of Supercomputing '98*.

Lawson, B. G. and E. Smirni. 2002. Multiple-Queue Backfilling Scheduling with Priorities and Reservations of Parallel Systems. In *Proceedings of 8th Job Scheduling Strategies for Parallel Processing*.

MPI: Message Passing Interface. Available via <http://www.mpi-forum.org>.

Rajaei, H., and M. Dadfar. 2005. Job Scheduling in a Cluster Computing. In *Proceedings of the 2005 American Society for Engineering Education Annual Conference*. ASEE.

Rajaei, H., and M. Dadfar. 2006. Comparison of Backfilling Algorithms for Job Scheduling in Distributed Memory Parallel System. In *Proceedings of the 2006 American Society for Engineering Education Annual Conference*. ASEE.

Shmueli, E. and D. G. Feitelson. 2003. Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling. *Job Scheduling Strategies for Parallel Processing (JSSPP)*. Lecture Notes in Computer Science, 2862, Springer-Verlag, pp. 228–251.

Srinivasan, S., R. Kettimuthu, V. Subramani, and P. Sadayappan. 2002. Characterization of Backfilling Strategies for Parallel Job Scheduling. In *Proceedings of IEEE International Conference on Parallel Processing Workshops,* pages 514–519.

Talby, D., and D. G. Feitelson. 1999. Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack-based Backfilling. In *Proceedings of the 13th IEEE International Parallel Processing Symposium, pp. 513.*

Uwe, S., and Y. Ramin. 1998. Improving First-Come-First-Serve Job Scheduling by Gang Scheduling. *Job Scheduling Strategies for Parallel Processing* (JSSPP) pp.180-198.

Wiseman, Y., and D. G. Feitelson. 2003. Paired Gang Scheduling. *IEEE Transactions on Parallel and Distributed Systems*. 14(6), pp. 581-592.

Zhang, Y., H. Franke, J. Moreira,. and A. Sivasubramaniam. 2003. An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration. *IEEE Transactions on Parallel and Distributed Systems*. 14(3), pp. 236-247.

AUTHOR BIOGRAPHIES

HASSAN RAJAEI is an Associate Professor of Computer Science at Bowling Green State University. His research interests include computer simulation, distributed and parallel simulation, performance evaluation of communication networks, wireless communications, distributed and parallel processing. Dr. Rajaei received his Ph.D. from Royal Institute of Technologies, KTH, Stockholm, Sweden and he holds an MSEE from Univ. of Utah. His e-mail address is <rajaei@cs.bgsu.edu> and his Web address is <http://www.cs.bgsu.edu/rajaei/>.

MOHAMMAD B. DADFAR is an Associate Professor in the Computer Science Department at Bowling Green State University. His research interests include Computer Extension and Analysis of Perturbation Series, Scheduling Algorithms, and Computers in Education. He currently teaches undergraduate and graduate courses in data communications, operating systems, and computer algorithms. He is a member of ACM and ASEE. His e-mail address is <dadarf@cs.bgsu.edu> and his Web address is <http://www.cs.bgsu.edu/dadfar/>.

PANKAJ JOSHI was a graduate student of Computer Science at Bowling Green Sate University. He graduated in 2005 and joined the industry.