JOB SCHEDULING IN CLUSTER COMPUTING AS A STUDENT PROJECT

Hassan Rajaei and Mohammad B. Dadfar
Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403
{rajaei, dadfar}@cs.bgsu.edu

Abstract

Cluster computing is widespread today by utilizing commercial-off-the-shelf (COTS) products. What concerns most users, however, is the long waiting time for their jobs to be assigned to the available computing nodes. In this paper, we present a study of job scheduling in cluster computing by employing several scheduling policies for the distributed environment. We analyze the behavior of the targeted scheduling policies as well as the jobs arriving into the system. The project provided excellent opportunities for students' learning on advance topics in operating systems and distributed computing. Scheduling for a distributed system is regarded as a non-trivial task. However, our methods challenges students to look beyond the scenes and realize the importance of fairness in complex distributed computing.

Job scheduling in distributed systems often constitutes a challenge. Traditional measures such as job-length and requested number of computing nodes do not suffice. Communication delays and synchronization overhead frequently overshadow the performance and hence appear as key issues for multiprocessor utilization. Consequently, performance of the entire system may degenerate if the job scheduler is not fine-tuned. User jobs may risk waiting long in the queue before getting the requested number of processors. This paper investigates and evaluates variable partitioning schemes for job scheduling on distributed-memory parallel systems. Further, it suggests how the methods could be used as student projects for an advanced operating systems or parallel programming course.

Introduction

High Performance Computing (HPC) nowadays can easily be achieved with clusters of PCs connected through a high-speed switch on a high-speed network. Such a tool provides excellent opportunities to explore numerous projects for educational as well as research purposes in computer science. For this reason, we have installed a Beowulf Cluster[1,2,3,4,5] with 16 compute nodes in our computing lab in order to engage our students with exciting projects in courses such as Operating Systems, Data Communication, Parallel Programming, Distributed Simulation, Algorithms, Database Management, and several others.

The Message-Passing Interface (MPI) library[4,6] and a fairly simple resource manager transform the cluster into a distributed-memory parallel system. The job scheduler module of the parallel machine has a vital role on the utilization of the distributed resources. The student project aims at studying feasible approaches to job scheduling on the parallel system. Further, it targets improving the overall performance of our Beowulf system by modifying the existing scheduler.

The scheduling of parallel applications on a distributed-memory parallel system often occurs by granting each job the requested number of processors for its entire run time. This approach is referred to as variable partitioning which frequently utilizes non-preemptive batch scheduling[7]. That is, once a job acquires the requested number of compute nodes, it continues until the job completes or some error forces the system to abort the faulty job. Consequently, most parallel programs restrict their I/O bursts to the beginning and the end of the program in order to avoid significant performance penalties.

Another scheduling approach is dynamic partitioning. This scheme suggests partial allocation of requested nodes for a parallel job. Even though this approach could assist certain jobs to start processing their task, the scheduling method is not widely used because of practical limitations. For example, consider a job where it needs all its requested nodes in order to start processing the parallel tasks. In this case, allocating fewer compute nodes than the requested numbers will clearly contribute to system deficiencies since the allocated nodes are not going to be used until this job receives the rest of its requested nodes.

We divide the project into several phases. In each phase, one to three students investigate how to minimize waiting time of the jobs in the queue while allowing other projects have a scheduling policy that suits their experiment and research.

In this paper we focus on the variable partitioning schemes[8,9,10]. The simplest method is to prioritize waiting jobs based on a preset policy, such as the arrival time, the job length, and the estimated wall-clock time. When resources for the high priority job are not available or there are no higher priority jobs in the queue, then the lower priority jobs are allowed to acquire the available computing nodes. This approach seems to have the advantage of better utilizing the system resources. However, a closer look reveals several pitfalls of this method. Examples include: jobs could starve; no guarantee can be made to the user as to when a job is likely to be executed; and the high priority jobs may risk starving.

The student project aims at investigating potential deficiencies and tries to provide alternative solutions for variable partitioning schemes. The goals are twofold: to provide the students with challenging real-world problems, and to improve our existing scheduler in our clustered-base parallel system. To overcome some of the drawbacks mentioned above, this paper selects approaches such as reservations for jobs and a backfilling technique to increase system utilization.

The three scheduling algorithms in the framework of variable partitioning that we have focused on are: *Non-FCFS*, *Aggressive Backfilling*, and *Conservative Backfilling* [8, 9]. In the next sections we briefly describe each of these algorithms. We then discuss the implementation issues. Furthermore, the simulation results are provided. Finally, the future work and the concluding remarks are presented.

Non-FCFS Scheduling Algorithm

One simple scheme in variable partitioning is to prioritize the jobs in the waiting queue based on a preset policy such as the requested number of processors or the estimated wall-clock time in addition to the arrival time. The resource manager then tries to allocate compute nodes to the waiting jobs in the order inserted in the queue. When resources for the job at the head of the line with the highest priority are not available then other jobs in the queue with lower priority can obtain the available resources. This approach has three pitfalls: jobs can starve; no guarantee is made to the user as to when a job is likely to be executed; and there will be no real priority since the execution of high priority jobs could be delayed and subject them to starvation. However, most schedulers that use this simple approach employ a simple starvation prevention policy by enforcing an upper bound for waiting. These systems normally use two priority levels and a certain time limit, for example 12 or 24 hours, for a job to be in the Non-FCFS waiting queue. After this time limit the priority is increased and the FCFS policy is enforced. Another way to prevent starvation would be to allow only a certain number of lower priority jobs to jump over a queued job. The OpenPBS Torque [2], which is incorporated in numerous clusters, employs such a policy. Starvation can be prevented at the cost of utilization.

Example of Job Scheduling Using Non-FCFS Scheduling Algorithm

Consider a system with 16 nodes and a 24 hour wait limit to prevent starvation. Table 1 provides an example of a set of jobs in the system and Figure 1 illustrates a snapshot of the scheduler.

Table 1: Status of current jobs in the system before 24 hours wait limit.

Job ID	Nodes Needed	Time unit	Status
Job1	6	3	running
Job2	6	2	running
Job3	12	1	queued less than 24 hours
Job4	5	3	new arrival – jumps over Job3
Job5	4	2	new arrival – jumps over Job3

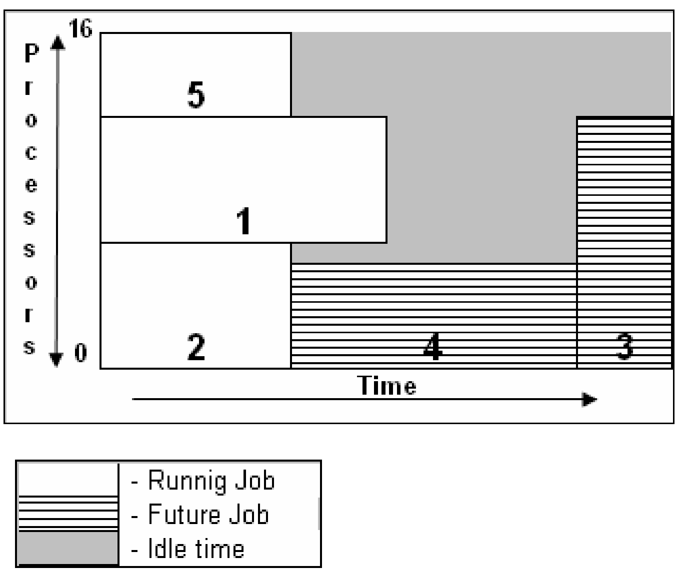


Figure 1: Non-FCFS without wait limit.

Selection of the jobs for execution is based on the number of requested processors. When there is a tie between the requests, then the jobs' execution time are considered. As illustrated in Table 1, Job1 and Job2 are running. Job1 has 3 time units to complete and Job2 has 2 time units to finish its task. Job3 is queued because there are not enough nodes available. Assume that Job4 and Job5 have just arrived in the system. Job5 gets immediate allocation since there are 4 free nodes left idle in the system. When Job2 terminates, Job4 will be able to run. As illustrated, the two new jobs move ahead of the waiting Job3 and acquire the resources. The fairness is poor in this scheme. Further, if more new small jobs arrive, then the large jobs will starve assuming there was no upper bound for waiting.

Now, consider the same example, but assume that Job3 has been queued in the system more than 24 hours, as illustrated in Table 2. Figure 2 demonstrates the snapshot of the system when waiting time exceeds the 24 hour limit.

Table 2: Status of current jobs in the system after 24 hour wait limit.

Job ID	Nodes Needed	Time unit	Status
Job1	6	3	running
Job2	6	2	running
Job3	12	1	queued more than 24 hours
Job4	5	3	new arrival – queued
Job5	4	2	new arrival – queued

Figure 2 shows arrival of Job4 and Job5 in the system. Job3 has already been waiting for more than 24 hours exceeding the wait time limit. The scheduler inserts Job4 and Job5 after Job3 in the queue. Even though allocation of nodes for Job5 would not delay the execution of Job3, the scheduler just queues all the incoming jobs in the arriving order until the starved job (i.e. Job3) has acquired the requested resources. This simple technique would clearly lead to low system utilization, however, it will prevent starvation.

Aggressive Backfilling Algorithm

This scheme requires the user to provide an estimated runtime in order to overcome the deficiency problem of Non-FCFS algorithm. With this added information the scheduler makes a reservation for the queued job. Then, it scans through the waiting queue to find a

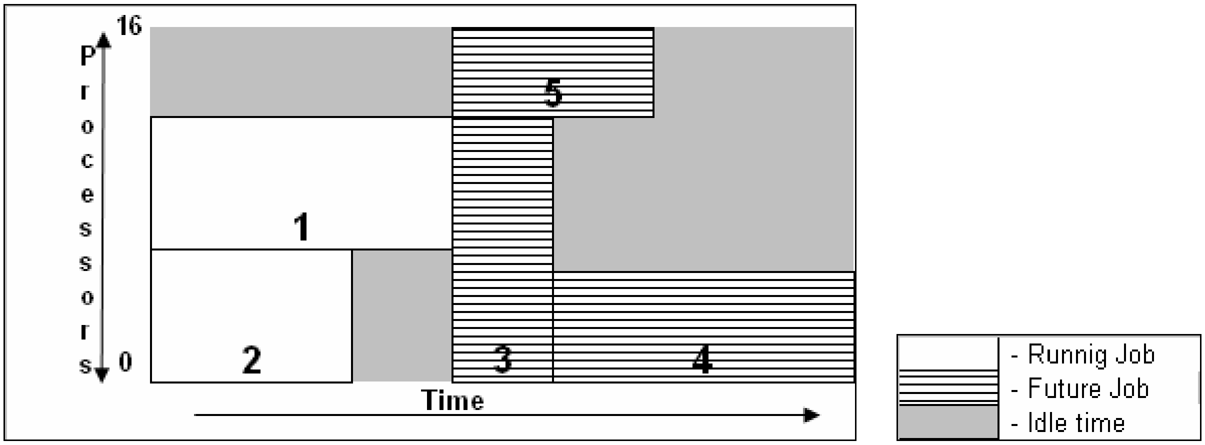


Figure 2: Illustration of waiting time limit on Non-FCFS.

smaller job which can be run ahead of the reserved job without imposing any further delay for the reserved jobs. This algorithm solves the starvation problem as well as improving system utilization by using a backfilling technique. That is, a job that does not risk delaying the reserved job is allowed to execute prior to the reserved job. The drawback of this technique is that it cannot make any guarantee about the response time of the user job at the time of job submission. Further, the user estimate may not be accurate. Early terminations and exceeding the estimated runtime have to be dealt with. Early termination may not cause serious problems whereas exceeding the estimated runtime may generate numerous problems.

Another issue is how to handle high priority job arrival. If a new job has a higher priority than the reserved job in the queue, then the system has two choices: either preempt the existing reservation and schedule the new job, or make another reservation for the new job immediately after the current reservation without preempting the job. There is no simple solution for this case. Choosing the former may result in starvation again as the higher priority jobs may continue to arrive. Choosing the latter approach is not fair for the high priority jobs as their requests could be delayed and hence risks not to be scheduled on time.

Example of Job Scheduling using Aggressive Backfilling Algorithm

Consider again the system of 16 nodes. Table 3 shows a set of jobs in the system ordered based on their arrival.

Table 3: Status of current jobs in the system for a backfilled queue.

Job ID	Nodes Needed	Time unit	Status
Job1	6	3	running
Job2	6	1	running
Job3	12	1	queued
Job4	14	1	queued
Job5	4	2	new arrival – backfilled
Job6	4	3	new arrival – backfilled

Job3 is the first queued job so it has a reservation in the system. Job4 is queued behind Job3. When Job5 and Job6 arrive, the system attempts to backfill these jobs. Job5 can be backfilled and be scheduled immediately. Job6 is queued. This case is shown in Figure 3a for the system after arrival of Job4 and Job5. When Job2 has terminated, its 6 nodes become available for 2 time units before Job1 terminates. Since Job6 requires 3 time units the system cannot schedule it. Job6 is scheduled after the termination of Job5. This case is demonstrated in Figure 3b, after Job5 has terminated. At time 3, Job3 starts its execution.

Now that Job3 has been removed from the ready queue, Job4 becomes the first job in the queue so the system makes a reservation for it. Figure 3c shows a snapshot of the system after Job3 starts execution. Figure 3d illustrates the overall snapshot. This example also demonstrates that the queued jobs (except the first one) can be delayed because of backfilling. This is a drawback which needs further analysis when such a scheme is used.

Conservative Backfilling Algorithm

In the Aggressive Backfilling algorithm only one reservation is made for the job in front of the queue. This could delay unnecessary execution of a job even though enough resources may exist to allocate for that job. The situation can be improved by allowing the scheduler to take a further step in backfilling. In Conservative Backfilling, all jobs get their own reservations when they are submitted. Therefore this algorithm can guarantee execution time when a new job is submitted. However, the algorithm works only for the First-Come First-Served (FCFS) priority policy. As jobs arrive in the system, the scheduler makes a reservation for them and provides a guaranteed execution time for each arriving job based on the estimated times provided by the users. When a job with higher priority arrives, the system cannot reshuffle its current reservations to provide the higher priority job a reservation ahead of the existing ones for the previous queued jobs. The reason is simple since any rearrangement would lead to existing jobs not being executed at their guaranteed times. A system using Conservative Backfilling with guaranteed execution time can only have FCFS priority.

Early job terminations lead to vacancies in the system. In order to make efficient use of these vacancies the algorithm must reschedule the existing reservations for queued jobs. However, keeping in mind that the reservations cannot be reshuffled as that may lead to not executing the jobs at their guaranteed times, we can only compress the existing reservation schedule so that it runs at an earlier time. This however may lead to an unfair scheduling policy.

Example of Job Scheduling Using Conservative Backfilling Algorithm

Table 4 shows a set of jobs in the system. As is the case for Conservative Backfilling, all jobs in the table with status as "queued" have a reservation. Job3 has its reservation after the termination of Job1. Job4 has its reservation after the termination of Job3. The system backfills Job5 as it does not delay the jobs that have reservation. Since Job6 cannot be backfilled, the system makes a reservation for Job6 after the termination of Job4. A snapshot of the system is illustrated in Figure 4.

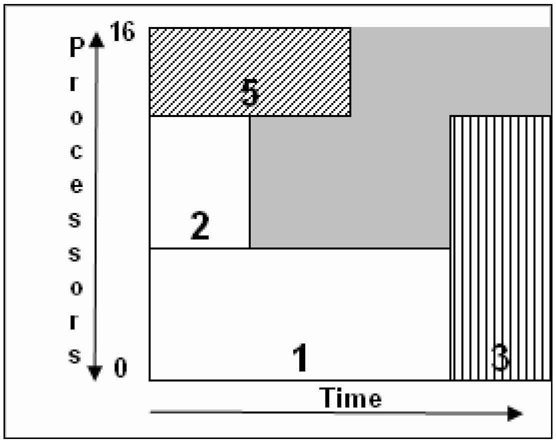
Table 4: Status of current jobs for an improved backfilled queue.

Job ID	Nodes Needed	Time unit	Status
Job1	6	3	running
Job2	6	1	running
Job3	12	1	queued
Job4	14	1	queued
Job5	4	2	new arrival – backfilled
Job6	4	3	new arrival – queued

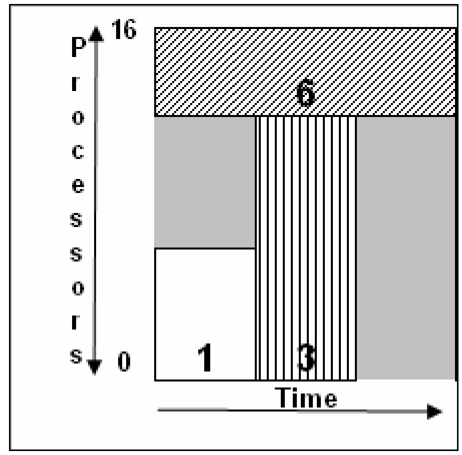
Implementation Issues

We have simulated the three algorithms discussed above. This section clarifies some issues related to implementation. More details can be found in[10].

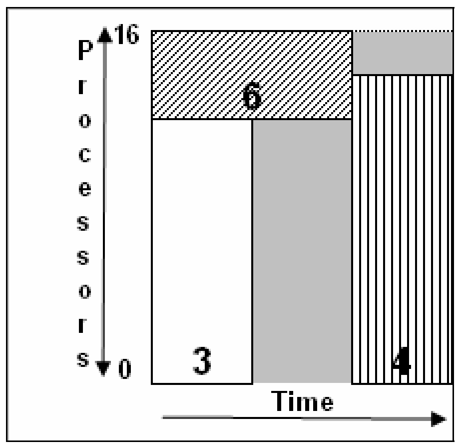
In a simulation study, the quality of input data plays a vital role in determining the significance of the simulation results. To safeguard our simulation, we looked at some observations made by researchers in the field of distributed-memory parallel systems [8, 9]. These studies indicate small jobs (i.e. requiring few compute nodes) are more frequent than large ones (i.e. requiring large number of compute nodes). Further, jobs with short runtimes are more frequent than jobs with long runtimes. Based on these observations, we generated the input data for 16 compute nodes. The main characteristics



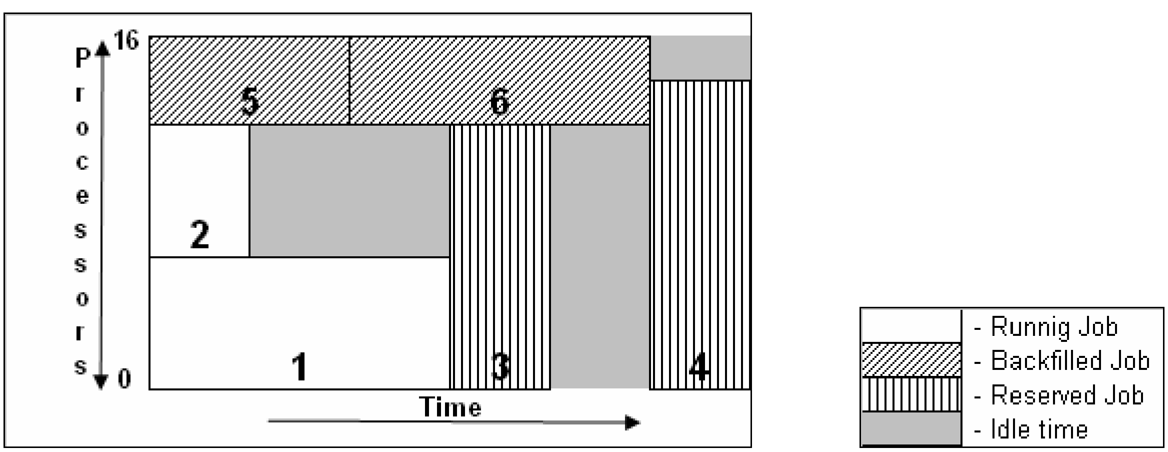
(a): System after the arrival of Job4 & Job5.



(b): System after Job5 terminated.



(c): System after Job3 starts execution.



(d): Overall job scheduling order.

Figure 3: Illustration of the Aggressive Backfilling algorithm based on their arrival.

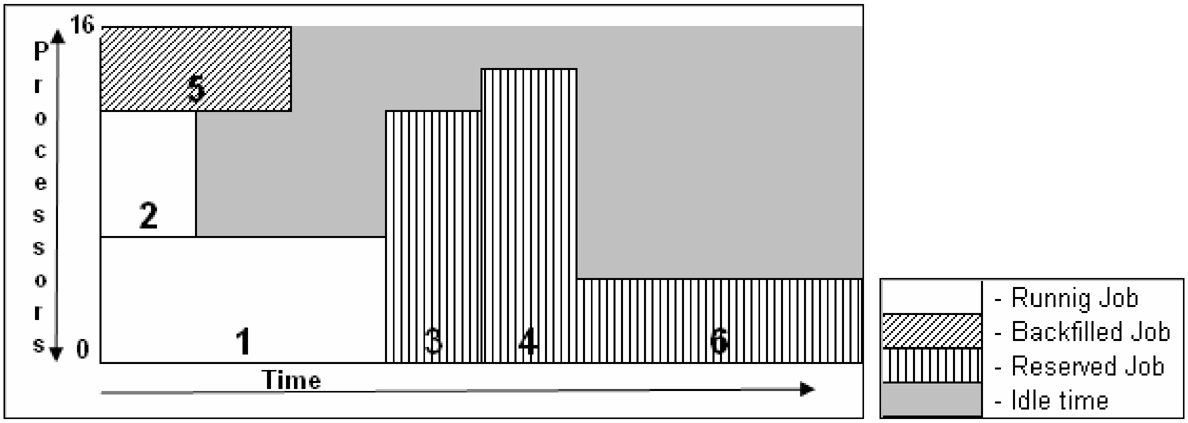


Figure 4: Snapshot of a Conservative Backfilling algorithm for jobs in Table 4.

of the input data are:

1. 30% small jobs: 1-5 nodes, 40% medium jobs: 6-10 nodes, 30% large jobs: 11-16 nodes.

2. 80% of jobs with simulated runtime between 90 seconds and 9 hours, 20% of jobs with simulated runtime between 9 hours and 12 hours. We use a minimum of 90 seconds runtime which also includes overhead of parallel execution.

3. A simulated runtime may be anywhere between 10% to 100% of the estimated runtime.

The input data is kept in a workload file and is transferred to the job profiles in the simulator. The supplied information for each job includes: process-ID, number of requested nodes, estimated runtime in seconds, simulated runtime in seconds, and arrival time in seconds. The arrival time is used to sort jobs in the ready queue prior to their scheduling or reservation of nodes.

For the Non-FCFS algorithm, a 24 hour limit is used to determine starved jobs in the system. The backfilling algorithms employ the first-fit technique. This method looks for the first queued job that can be used for backfilling. It would be interesting to investigate whether other methods such as best-fit could have any impact on the performance.

The simulator utilizes two queues: event queue and ready queue. The event queue keeps track of the arrivals and departures of jobs and generates events for the respective type. The simulation clock advances when the system processes an event. The ready queue on the other hand holds all arrival jobs for which their requested nodes are not yet granted. This queue is used by the algorithms to schedule the jobs. Once a job arrives in the system, its profile will be moved from the event queue to the ready queue. When a job is scheduled, its profile will be moved back from the ready queue to the event queue. A departure event will remove the profile of the terminated job. Both queues are implemented as vectors.

Simulation Results

Table 5 summarizes results of the simulation. The workload file that was used for the input data was generated by using the specifications described previously. To facilitate interpretation of the obtained results we assume the simulated runtime of a job be the same as a perfect estimation time provided by the user. Table 6 summarizes the results of the second simulation. All illustrated times in the tables are simulated times and are in seconds.

We also used a set of input data which has been used in a simulation carried out at Ames Laboratory [7]. For this set of data, Tables 5 and 6 summarize the results of our simulation.

The results shown in Table 5 through Table 8 depict that for Backfilling algorithms utilization is better than the Non-FCFS algorithm. Both Aggressive Backfilling and Conservative Backfilling have almost the same utilization. However, for both algorithms the average wait time is higher than the one from Non-FCFS. The highest response time (i.e. the job that waited the most in the system) is lowest in the Aggressive Backfilling algorithm.

If we consider the case when the arrival job has a perfect estimate, then the utilization becomes the same in both Aggressive Backfilling and Conservative Backfilling algorithms. With perfect estimated runtime, utilization increases marginally. For example, in the case of Conservative Backfilling, utilization increases by 2% from 87.09% to 89.20% (see Table 5 and Table 6).

The simulation results also indicate that more jobs are executed in their order of arrival for both Backfilling algorithms, but that is not the case with Non-FCFS where many jobs that arrived late could finish ahead of the jobs that arrived before them. This shows that the Aggressive Backfilling and the Conservative

Table 5: Results of simulation for generated job mix.

Algorithm Name	Average Execution Time (sec)	Average Wait Time	% System Utilization	Highest Response Time
Non-FCFS	308.37	101.04	85.49	296.36
Aggressive Backfilling	301.37	107.52	87.48	284.88
Conservative Backfilling	302.72	107.24	87.09	290.55

Table 6: Results of simulation for jobs submitting with prefect estimate.

Algorithm Name	Average Execution Time (sec)	Average Wait Time	% System Utilization	Highest Response Time
Aggressive Backfilling	764.38	262.55	89.36	747.08
Conservative Backfilling	765.75	280.52	89.20	743.59

Table 7: Results for the simulation job mix available using different input data.

Algorithm Name	Average Execution Time (sec)	Average Wait Time	% System Utilization	Highest Response Time
Non-FCFS	68.55	19.92	77.99	62.73
Aggressive Backfilling	65	20.08	82.25	59.18
Conservative Backfillir	65.17	20.14	82.04	59.35

Table 8: Results for the simulation job mix using different input data with prefect estimate.

Algorithm Name	Average Execution Time (sec)	Average Wait Time	% System Utilization	Highest Response Time
Aggressive Backfilling	80.02	25.23	83.14	74.14
Conservative Backfilling	79.17	25.42	84.04	73.29

Backfilling algorithms follow the priority policy more closely than the Non-FCFS algorithm. Furthermore, the simulation results show that the Backfilling algorithms are able to distribute the wait time among the jobs more evenly instead of just making a few jobs wait indefinitely.

Future Work

For the next phases, we would like to expand the simulation using variable nodes and launch numerous actual jobs. These jobs could be managed by a mixture of simulated and experimental scheduler to provide us with better measured data to further study and analyzing them. We would like also to study additional scheduling algorithms and provide experimental models for students to use in courses such as Operation Systems.

One job scheduling that carries a high research potential is dynamic partitioning. In this approach, a parallel job does not always need to get its requested number of processors for its entire run time. Two such techniques that are still widely under research are Dynamic Co-scheduling and Gang scheduling[11, 12]. Both of these approaches try to allocate more than one process to a node and time-share that node between the processes. The difference is that Gang scheduling is similar to round robin scheduling in which the system switches to a new sets of jobs after a fixed time-slot. Dynamic co-scheduling, on the other hand, uses bob arrivals to trigger execution. The dynamic co-scheduling can start a job even when all of its requested nodes are not available. Simulation can be done on these two approaches to closely study their performance.

Utilization of compute nodes tends to decrease in very large parallel systems. In our future work, we can model workloads for very large systems and study performance of various scheduling algorithms on such systems.

Moab Workload Manager is a cluster scheduler that is compatible with the Torque; an OpenPBS based cluster resource manager [7]. The Moab scheduler has several configuration settings which can provide the administrator with a greater flexibility in changing the scheduling algorithm to suit some specific needs of the system. This scheduler can be configured to work as both an Aggressive Backfilling scheduler and a Conservative Backfilling scheduler. In future, we would like to install the Moab scheduler in a Beowulf system and fine-tune it to suit our needs.

Concluding Remarks

The first phase of this project has provided significant insight on how our current work could continue in several directions. Our primary objectives are still valid. That is, providing exciting projects for students in a multiprocessor environment, as well as improving the current job scheduler for our Beowulf cluster. The preliminary results of this study confirm that cluster scheduling could furnish several interesting student projects for educational purposes, as well as providing several challenging topics in cutting edge research.

Furthermore, our simulation results advocate that Backfilling algorithms produce better utilization than Non-FCFS algorithm. The average wait time for a job, however, is higher in the Backfilling algorithms than the one in the

Non-FCFS algorithm. This could nonetheless be a result of distributing wait time among many jobs. Our future work will pursue to better address this issue. The simulation also shows that the two Backfilling algorithms perform almost identically. Between the two, Aggressive Backfilling is less complex and more flexible for the purpose of setting priority policy. The Conservative Backfilling algorithm can be used in systems that desire to have scheduling policy set strictly based on FCFS. Conservative Backfilling will be able to give the users a guaranteed response time.

Bibliography

1. http://www.beowulf.org/: Beowulf Project Overview

2. http://www.supercluster.org/: Center for HPC Cluster Resource Management and Scheduling

3. http://www.cacr.caltech.edu/beowulf/tutorial/ Tutorial.html: How to Build a Beowulf: a Tutorial.

4. Gropp, William, Lusk, Ewing and Sterling, Thomas, Beowulf Cluster Computing with Linux, Second Edition, ISBN 0-262-69292-9, 2003.

5. Ehammer, Max, Roeck, Harald, and Rajaei, Hassan, "User Guide for the Beowulf P4 Cluster" Department of Computer Science, Bowling Green State University, Bowling Green, OH 43403,USA, July 2004.

6. Gropp, William, Lusk, Ewing and Sterling, Thomas, Using MPI, Portable Parallel Programming with Message-Passing Interface, Second Edition, ISBN 0-262-57132-3, 2003.

7. Bode, Brett, Halstead, David M., Kendall, Ricky and Lei, Zhou "The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters". In Annual Technical Conference, USENIX, June 1999.

8. Mu'alem, Ahuva W. and Feitelson, Dror G. "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling". In IEEE trans. Parallel & Distributed Systems 12(6), pp. 529-543, June 2001.

9. Feitelson, Dror G. Packing schemes for gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, 2ndWorkshop on Job Scheduling Strategies for Parallel Processing (in IPPS '96), pages 89–110, Honolulu, Hawaii, April 16, 1996. Springer-Verlag. Published in Lecture Notes in Computer Science, volume 1162. ISBN 3-540-61864-3. Available from http://www.cs.huji.ac.il/~feit/parsched/p-96-6.ps.gz.

10. Alagusundaram, Kavitha "A Comparison of Common Processor Scheduling Algorithms for Distributed-Memory Parallel System", Department of Computer Science, Bowling Green State University, Bowling Green, OH 43403, USA, May 2004.

11. Talby, David, Feitelson, Dror G., "Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack Based Backfilling", Institute of Computer Science, The Hebrew University, 91904, Jerusalem, Israel.

12. Jette, Moe, "Gang Scheduling Timesharing on Parallel Computers", http://www.IInl.gov/asci/pse_trilab/sc98.summary.html

Biographical Information

Hassan Rajaei is an Associate Professor in the Computer Science Department 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 holds an MSEE from University of Utah.

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.