High Performance Computing Student Projects

Hassan Rajaei and Mohammad B. Dadfar
Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403

Abstract

Commodity High Performance Computing (HPC) platforms such as Beowulf Clusters provide excellent opportunities to engage students with challenging projects. Courses such as parallel programming, distributed systems, operating systems, and networking can benefit from the low-cost HPC platform. In this paper we report the results on series of student projects in an advanced operating systems course which jointly have contributed to a larger group project. Several students designed, implemented, and tested segments of manageable term projects contributing to the student learning in the advanced topic of high performance computing. We focused on job scheduling for a cluster of processors as the main topic, while pursuing other HPC-related areas such as parallel programming, load balancing, computer simulation, and performance analysis embedded in the theme.

In this paper we examine the following scheduling polices: FCFS (First-Come-First-Serve), Backfilling Algorithms (Aggressive, Conservative, Multiple Queue, Look-ahead), Co-scheduling, and Gang Scheduling. While most of the scheduling policies are batch, Gang Scheduling provides a timesharing approach to the multiprocessor system. Our results indicate that Gang scheduling offers an attractive solution to the drawbacks of batch scheduling. This is especially true with respect to the response time and overestimation of the processing time of the submitted jobs in the system.

Introduction

High performance computing offers an excellent vehicle to accelerate computational needs of scientific and engineering applications. This platform can easily be configured with clusters of PCs connected through a high-speed switch on a high-speed network. Such a tool provides exceptional opportunities to explore numerous projects for educational as well as research purposes. We have installed a Beowulf Cluster[1] with 16 compute-nodes in our computing lab, and have engaged our students with exciting projects in courses such as Operating Systems, Communication Networks, Parallel Programming, Distributed Simulation, Algorithms, Database Management, and several others. Within a short period of time, we have witnessed considerable increase in student projects in our HPC lab with several success stories[2,3]. Student interest and their reported success are growing. They are excited to work with advanced and practical problems which take them beyond the theory of their textbooks.

In this paper, we provide a comprehensive report about a series of studies on job scheduling in a distributed multiprocessor environment, which engaged several students in a three-year period. We also show how students can utilize and build upon the results of previous groups while laying the ground work and providing continuity for the future students. The incremental development method was particularly beneficial both for the students and the faculty. Students were well aware that their work would be used by others, and as a result, paid extra attention to the viability of their work which contributed to a great learning outcome. Integrating small projects into larger ones has twofold benefits for faculty: achieving the research objectives, and sharing the obtained results in the classroom. These studies encapsulated challenging HPC-related components such as parallel computing, load balancing, and distributed simulation.

Scheduling of parallel applications on 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[4, 5, 6]. That is, once a job gets hold of the requested number of compute-nodes, it continues execution until the job is completed or some error forces the system to abort the faulty job. Consequently, most parallel programs restrict their I/O bursts to the beginning and 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 help some jobs to start processing their task, the scheduling method is not widely used because of practical limitations. As an 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 requested can lead to system deficiencies since those allocated nodes can become idle until this job receives all of its requested nodes.

A third method regards use of co-scheduling of the tasks and timesharing processor powers among existing jobs in the system. Co-scheduling can be implicit or explicit. We use the latter method described as Gang Scheduling[7,8,9]. In this method, all the processes of a parallel job are assigned to a Gang of processors for execution. Context switching is coordinated across the nodes such that all processes are executed and preempted at a fixed interval. Gang scheduling favors short jobs.

In our studies, we looked into the batch and co-scheduling policies with Backfilling Algorithms using different flavors such as Aggressive, Conservative, Multiple Queue, and Look-ahead schemes.

The remainder of this paper is organized as follows: the next section provides some challenging HPC topics suitable for student projects. Then it describes job scheduling problem in HPC platforms as well as details of the methods used in the studies. In the next section simulation techniques for the employed policies are described. Followed by a section that describes implementation issues, and then provides the obtained results and analysis. The final two sections outline future work and then provide concluding remarks.

HPC Stimulating Topics

High performance computing opens the doors for solving numerous fascinating scientific and engineering problems. However, several items should be considered at the same time. In order for a HPC task to obtain the desired answer, the programmer needs first to be familiar with *parallel programming* and *load balancing* issues. After that, the programmer needs to model the application domain problem in form of a *parallel program*. Knowledge of *load balancing*, *networking*, *operating systems*, and *parallel processing* facilitate the goal of reaching the desired results. On top of this list, we have added two more items: the application domain of *job scheduling in multiprocessor* environment, and *simulation* of the job scheduler in the multiprocessor system. These topics often stimulate senior and graduate students who desire to put their theoretical knowledge into practice.

We target an advanced operating systems course since in such a course students often learn about concurrent processes, communication and synchronization between processes, as well as task scheduling and policies. We train the students with parallel programming, load balancing, and simulation issues with simple examples and benchmarks.

For parallel programming purpose, the Message-Passing Interface[10,11] (MPI) library was used as part of the Beowulf cluster. MPI uses the master-slave paradigm similar to parent-child method of the Unix fork command. Students are well aware of this method and are excited to experiment with the execution of

concurrent processing on multiprocessors. As a warm-up exercise, we assign students the matrix multiplication benchmark problem. In this problem, we assume that we have two large matrices A and B to be multiplied and the result would be stored in matrix C. By varying the dimension of matrices A and B, as well as the number of processors to obtain the result, we will achieve a well-designed exercise where the students will observe the following:

- writing a simple parallel program to be executed on a Beowulf cluster
- partitioning the task between the existing number of processes and processors
- scheduling the task of multiplication to processors
- choice of static vs. dynamic task assignment
- impacts of method of assigning tasks to processors
- impacts of load balancing to the execution of parallel task
- performance of parallel execution and speedup
- inter-process communications with *send, receive, scatter, gather, and broadcas*t
- synchronization issues such as barrier, blocking and non-blocking send and receive
- understanding the ratio of computation vs. communication, and
- debugging issues in parallel execution.

The matrix multiplication benchmark is fairly straightforward; nevertheless it provides an excellent training for students and prepares them to perform their domain application task which in our case is scheduling of parallel programs arriving in multiprocessor environment such as a Beowulf Cluster.

Job Scheduling Problem

Job scheduling on distributed-memory multiprocessors is a challenging task. Traditional factors such as job length to allocate the requested resources do not suffice. Communication delays and synchronization overhead which are normally in the user domain could turn out to be key issues for multiprocessor utilization. If scheduling does not carefully address these and several other issues, the utilization of each processor in the distributed platform can end up comparatively lower than a single processor system. Such scenario should be avoided since high performance computing is all about performance. To better understand these issues, our studies looked into both batch orient non-preemptive schemes as well as preemptive timesharing polices. For both types, we used different flavors of backfilling methods such as conservative, aggressive, look-ahead, and multiple queues. In the following subsections we provide an overview of these algorithms.

Non-FCFS Batch Schemes

Strict First-Come First-Serve (FCFS) policies are seldom used for resource managers. Some sort of priority mechanism is often added to handle resource queues. One solution 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 the lower priority can obtain the available resources. This approach has three pitfalls: 1) jobs can starve, 2) no guarantee is made to the user as to when a job is likely to be executed, and 3) there is no real priority since high priority jobs can starve. However, most schedulers that use this approach employ a 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 (Portable Batch Scheduler) Torque[2], which is incorporated in numerous clusters, employs

such a policy. It is important to mention that starvation can be prevented at the cost of utilization.

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 the additional information this algorithm makes a first reservation scheduled for the queued job. Then, it scans through the waiting queue to find a 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 and improves system utilization by using 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 estimation may not be correct. Early terminations and exceeding the estimated runtime have to be dealt with. While early termination may not cause serious concerns, 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 reschedule for the new job, or make another reservation for the new job immediately after the current reservation without preempting it. There is no simple solution for this case. Choosing the former may result in starvation again as higher priority jobs may continue to arrive, whereas choosing the latter approach is not fair for the high priority jobs as their requests could be delayed and hence risking not be scheduled on time.

Conservative Backfilling Algorithm

In Aggressive Backfilling algorithm only one reservation is made for the job in front of the queue. This could delay execution of a job even though adequate resources may exist which can be allocated 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-Serve priority policy. As jobs arrive in the system, the scheduler makes 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 not executing existing jobs 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.

Look-ahead Backfilling Scheduling Algorithm

This algorithm tries to find the best packing possible for current composition of the queue, thus maximizing the utilization at every scheduling step[12,13]. The jobs are divided into two parts: *running* and *waiting* jobs. The jobs that are waiting may be either in the *Waiting Queue* or in the *Selected Queue*. The jobs in the *Selected Queue* are the ones selected for execution. All jobs have two attributes: *size* (number of requested processors) and estimated computing time remaining. The main task of

this algorithm is to select jobs from the *Waiting Queue* and improve system utilization.

The scheduler receives the incoming jobs from the job file specified by the user. When the scheduler starts, the simulation time is set to 0 and is incremented by 1 after each iteration. Incoming jobs get filed in the *Event Queue* according to their arrival time. The arrival time of the jobs in the *Event Queue* is compared with the CPU time. If they are equal, the jobs are moved to the *Waiting Queue*. Jobs in this queue are ordered by estimated execution time.

Multiple-Queue Backfilling Scheduling Algorithm

This algorithm is based on aggressive backfilling strategy. It continuously monitors the incoming jobs and rearranges them into different waiting queues. Rearrangement is necessary to reduce fragmentation of the resources and improve the utilization[4]. We define several waiting queues to separate the short jobs from the long ones. The scheduler orders the jobs according to their estimated execution time.

The system is divided into variable partitions and processors are equally distributed among the partitions. However, if a processor is idle in one partition then it can be used by a job in another partition. In effect, depending upon the work load of the jobs in the partitions, the processors are exchanged from one partition to another. In our simulation the algorithm uses four waiting queues instead of four actual partitions. Initially, each queue has equal number of processors assigned to it. We assume t_e represents the estimated execution time of a job and p_i represents the partition number where i = 1, 2, 3, 4. The jobs are classified into partitions p_1, p_2, p_3 and p_4 based on their execution times:

p_1 : $0 < t_e <= 100$
p_2 : $100 < t_e <= 1{,}000$
p_3 : $1{,}000 < t_e <= 10{,}0000$
p_4 : $10{,}000 < t_e$

Gang Scheduling

Gang scheduling refers to a policy where all the 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 resource utilization for parallel programs across the available compute nodes. We use a synchronization scheme which is coordinated by the master node (ParPar Scheduler or Score-D). Other options such as synchronized clocks (SHARE Scheduler IBM SP2) are also viable.

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 pre-empted, 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 not desirable.

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 computing nodes in any time slots are executed. The policy does not take into consideration 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.

Gang Scheduling with Backfilling

The jobs in the waiting queue are considered as per the look-ahead backfilling policy. The job at head of the waiting queue has the reservation of the computing 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 fair comparing to the greedy approach, but not as fair as the conservative backfilling approach.

Simulation Studies

Three methods were used for performance studies of the selected scheduling policies: Gantt-chart walk through, simulator using single machine, and simulators using Beowulf Cluster.

Gantt-chart walk through

As an example consider the case for aggressive backfilling algorithm in batch scheduling for a system having 16 compute nodes. Table 1 shows a set of jobs in the system ordered based on their arrival.

Table 1: 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

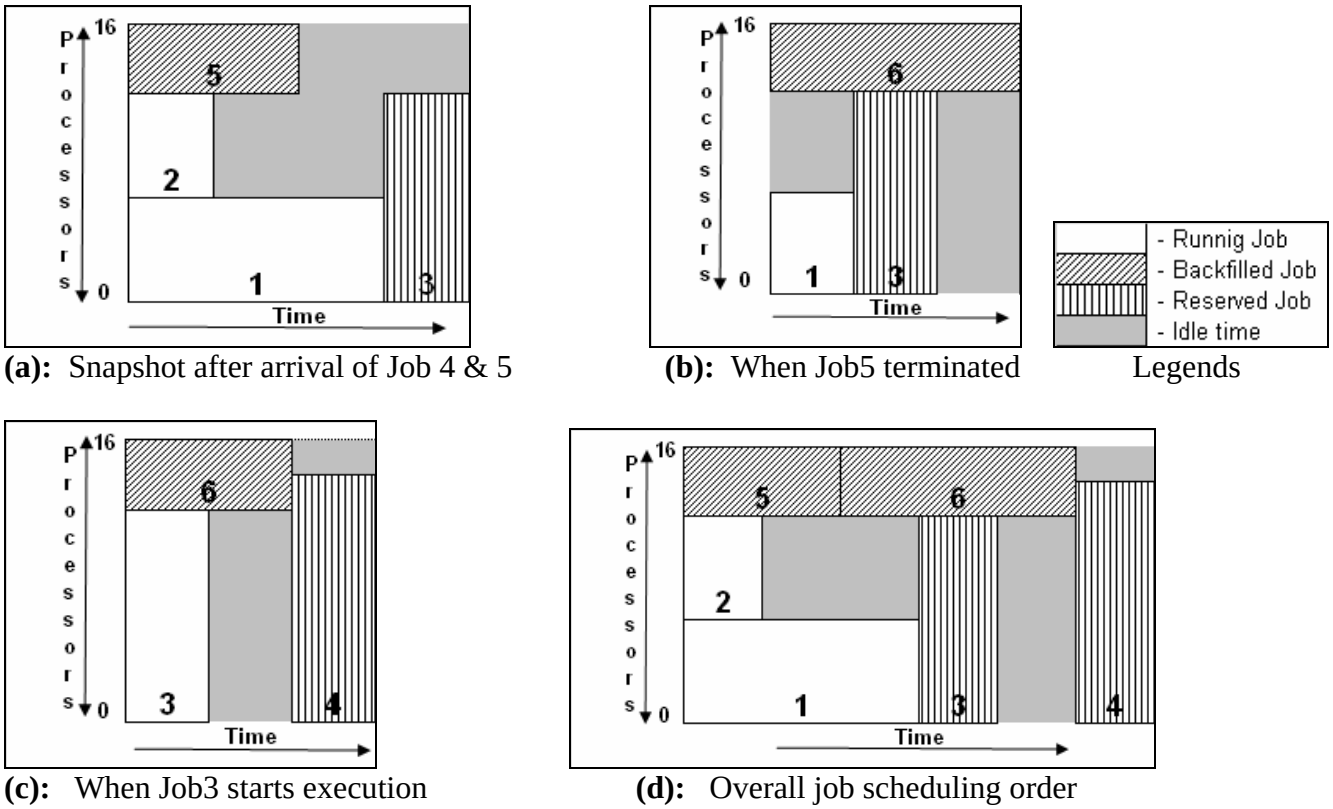
Job3 is the first queued job so it has a reservation in the system. Job4 is queued behind Job3. When Job5 and Job6 arrive, the system attempts to backfill the jobs. Job5 can be backfilled and scheduled immediately. Job6 is queued. This case is shown in Figure 1(a) for the system after arrival of Job4 and Job5. When Job2 is 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 1(b), 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 1(c) shows snapshot of the system after Job3 starts execution. Figure 1(d) illustrates the overall snapshot. This example also demonstrates the queued jobs (except the first one).

Simulation of Backfilling schemes

A base class simulator was developed to support different flavors of the backfilling algorithm. This simulator has a hierarchical view. The base simulator provides the base services needed for all favors. From this base class, other needed types of scheduler are derived. This is illustrated in Figure 2 with the *Basic Aggressive*, *Multiple-Queue*, and *Look-ahead*.

Look-ahead Backfilling Simulator

In Figure 3, a look-ahead scheduler is derived from the base class *Simulator*. It receives the incoming jobs from the job file specified by the user. When the scheduler starts, the simulation time is set to 0 and is incremented by 1 after each iteration. Incoming jobs get filed in the *Event Queue* according to their arrival time. The arrival time of the jobs in the *Event Queue* is compared with the CPU time. If they are equal, the jobs are moved to the *Waiting Queue*. Jobs in this queue are ordered by estimated execution time. Considering only the jobs in the *Waiting Queue*, the scheduler builds a matrix of size $(|WQ|+1) \times (n+1)$ where WQ is the *Waiting Queue* and n is the number of free processors in the system. Each cell of the matrix contains an integer value called *util* that holds the maximum achievable utilization at this time and a Boolean flag called *selected* that is set to true if it is chosen for execution. *Select Queue* selects all the jobs from *Waiting Queue* with the *selected* flag set to true. The utilization is calculated



(a): Snapshot after arrival of Job 4 & 5 **(b):** When Job5 terminated Legends

(c): When Job3 starts execution **(d):** Overall job scheduling order

Figure 1: Illustration of the Aggressive Backfilling algorithm based on the job arrivals.

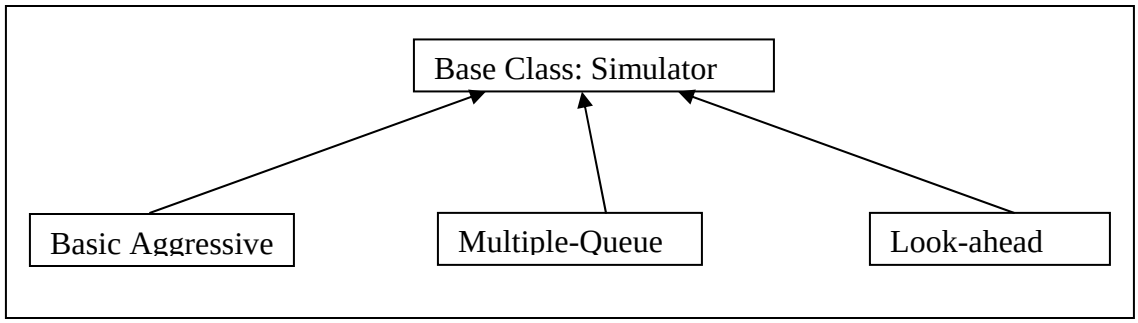


Figure 2: The class hierarchy of the simulators.

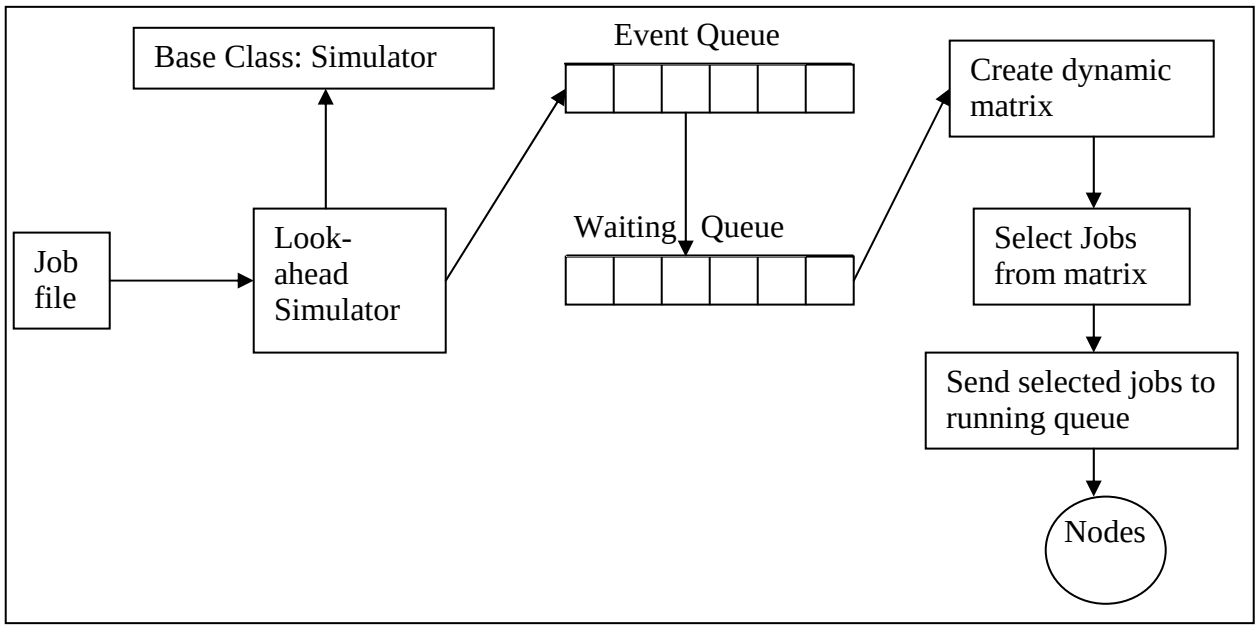


Figure 3: Overview of Look-ahead Backfilling Scheduler simulator.

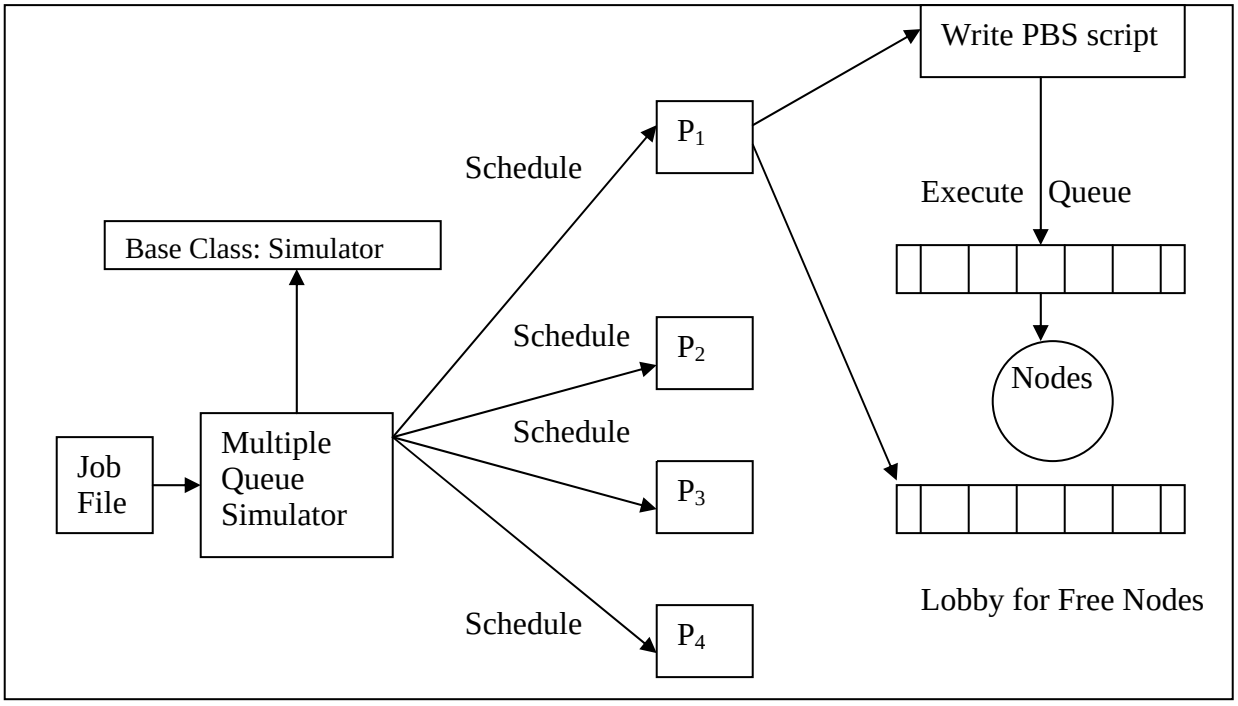


Figure 4: Overview of Multiple Queues Backfilling Scheduler simulator.

according to the number of computing nodes they have requested and what is currently available. The selected jobs then receive the number of nodes they have requested and start to execute.

Multiple-Queue Backfilling Simulator

In our implementation, Multiple-Queue Simulator is derived from the base class *Simulator*. When it receives input jobs it categorizes them into different waiting queues say P_1, P_2, P_3 and P_4 (Figure 4). The queues hold jobs based on their estimated execution time from 0 to 100, 101 to 1,000, and 1,001 to 10,000 and above 10,000 respectively. We use the MPI programming package[10], and have the first node considered as a *master* and the rest as the *worker* nodes. The scheduler program runs in the master node. It divides the computing nodes into groups of 4, 4, 4 and 3 for the queues P_1, P_2, P_3 and P_4 respectively (the master node does not participate in the computation).

Consider one of the queues in Figure 4, for example P_1. It holds jobs with execution time ranges of 0 to 100. They are ordered based on their estimated execution time and then the arrival time in case of ties. The scheduler starts checking the number of computing nodes requested by the first job. If there are enough free processors designated for queue P_1, then it records the PBS (Portable Batch Scheduler)[6] script and starts running that job. Otherwise, the job is sent to another queue called *Lobby for Free Nodes* where it waits for the free nodes before it can execute. If there is a job at this queue (Lobby for Free Nodes) the scheduler searches for free nodes from other queues (P_2, P_3, P_4) to check if the requested number of computing nodes could be granted. If the answer is yes, then resources will be allocated to that job to start execution. Otherwise, the job is transferred to Ready Queue (not shown in the figure). The scheduler uses the aggressive method to make reservation for the required number of nodes for that job. The same process is followed for jobs in the other partitions.

Simulation of Gang Scheduler

This simulator has different design and methods from the backfilling ones mentioned above. The architecture of the simulator is shown in Figure 5. 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[6] (PBS) 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.

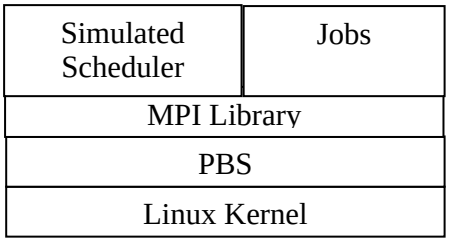


Figure 5: Gang scheduler simulator architecture.

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. 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.

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. 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. 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.

Implementation Issues

In this study, several backfilling scheduling policies as well as Gang Scheduling with Greedy Approach and Gang Scheduling with Look-ahead Backfilling Policy were simulated. This section provides some details of the implementations. Interested readers can contact the authors.

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/sec Ethernet switch; Gentoo Linux operating system; Batch System with PBS based Torque.

The Simulators

All simulators are written in C++. Those running on the cluster are 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 one-shot timer functionality. Subclasses need to override a method called *timerFunction*( ) which is invoked when the one-shot timer expires. The scheduler classes, derived from the Simulator class, make use of the timer to trigger events like global context switch and the start and the 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.

Results and Analysis

Batch processing with Backfilling

Experiments were carried out with a randomly generated workload. Further, the following parameters of interest were used: 1) Makespan: Total time to complete processing all jobs from a given pool representing utilization, 2) Wait Time: Length of time for a task waiting in the scheduling queue, 3) Number of requested compute-nodes, and 4) Estimated execution time.

The results for this part are shown in Figure 6 and Figure 7. Waiting time for the jobs in the Multiple Queue is more than Look-ahead backfilling algorithms and for the aggressive backfilling is smallest in comparison to the other two. Looking at the line graph of each algorithm in Figure 6 separately, it seems that all three algorithms have one thing in common: the execution of jobs do not depend on the arrival time. The jobs arriving late may execute before the other jobs that arrive before them, and hence, the algorithms are not fair. In case of look-ahead, the waiting time depends upon the utilization value of the job at that particular instant in time. The utilization value of each job is calculated by checking the number of requested processors and the number of available computing nodes at that time.

Requested Nodes versus Waiting Time

Figure 7 shows the waiting time of a job based on the number of compute nodes it needs. In the basic aggressive algorithm, jobs that request more nodes wait longer than jobs that request fewer nodes. For multiple queues, the jobs

requesting fewer nodes are executed before the jobs requesting more nodes in that queue. This figure suggests that the look-ahead backfilling

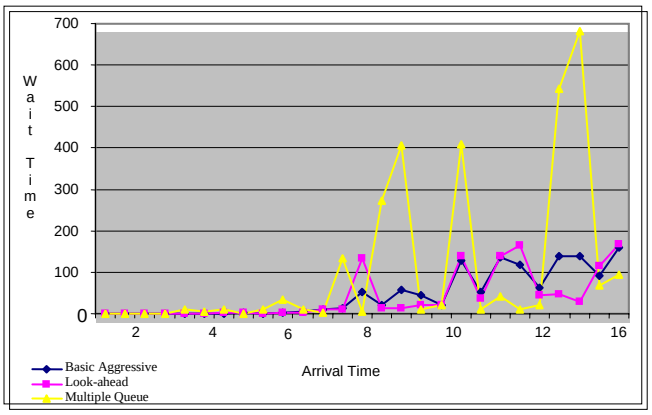


Figure 6: Arrival Time versus Wait Time of the three algorithms.

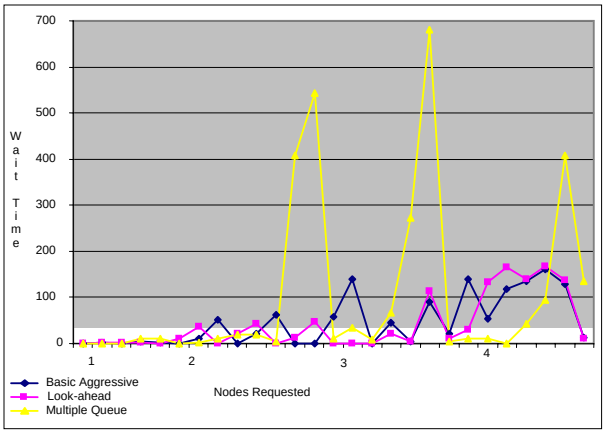


Figure 7: Nodes Requested versus Waiting Time.

algorithm provides better utilization. Further, jobs in Multiple Queue algorithm wait longer than the jobs in the other two backfilling algorithms.

Estimated Time versus Waiting Time

Normally, jobs with shorter estimated time are executed before jobs with larger estimated times. However, our results suggest that the look-ahead algorithm does not execute the jobs according to the estimated time of completion. In all three cases presented in our studies, Multiple Queue exhibits longer waiting time and look-ahead appears as a better choice.

Gang Scheduling and Backfilling

A 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 used in report. Figure 8 advocated that the Gang scheduling outperforms the backfill with look-ahead 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[14] (GS) category, the greedy approach seems superior to the backfilled one. Interestingly, GS with backfill tends to exhibit a behavior that is a compromise between backfill and GS with greedy approach. For fewer jobs, the GS with backfill coalesces with GS with greedy approach. This is because, as the number of jobs becomes smaller, time slots are readily available for most of them and neither the greedy nor the backfill policy effectively come 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.

As expected the average wait-time for the Gang scheduling is far less than that for the backfill as shown in Figure 8. In the case of backfill, the average response time increases more rapidly making it unsuitable for interactive jobs. The Gang scheduler performs better, 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.

Performance of the Gang scheduling exhibits noticeable improvement when the number of

time slots increases from 1 to 15 as suggested by Figure 9. With only one time slot, the Gang scheduler behaves like a batch scheduler. Further, as the number of time slots increases, the average wait time decreases significantly. As the system behaves like having virtual nodes equal to the number of slots multiply the number of actual nodes, more jobs can run without any delay, thus reducing the total and average wait time.

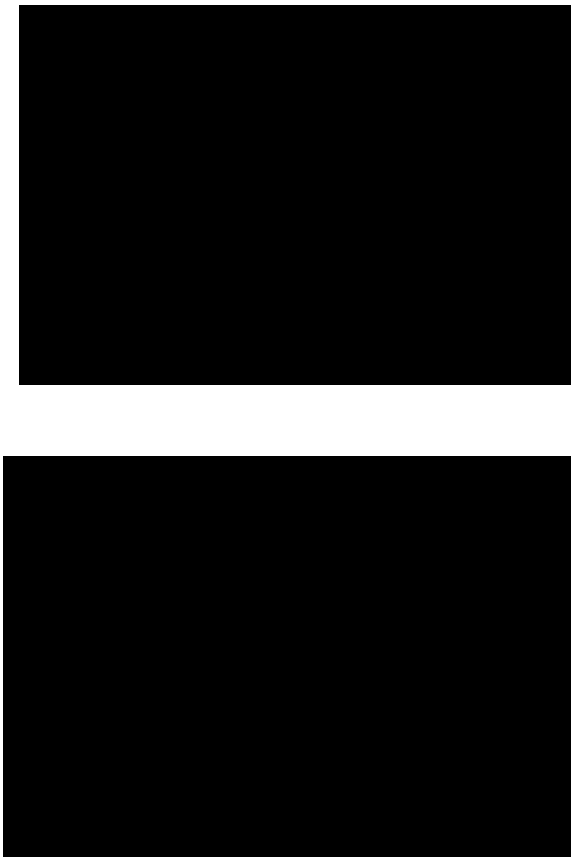


Figure 8: Makespan and Average Wait Time versus Number of Jobs

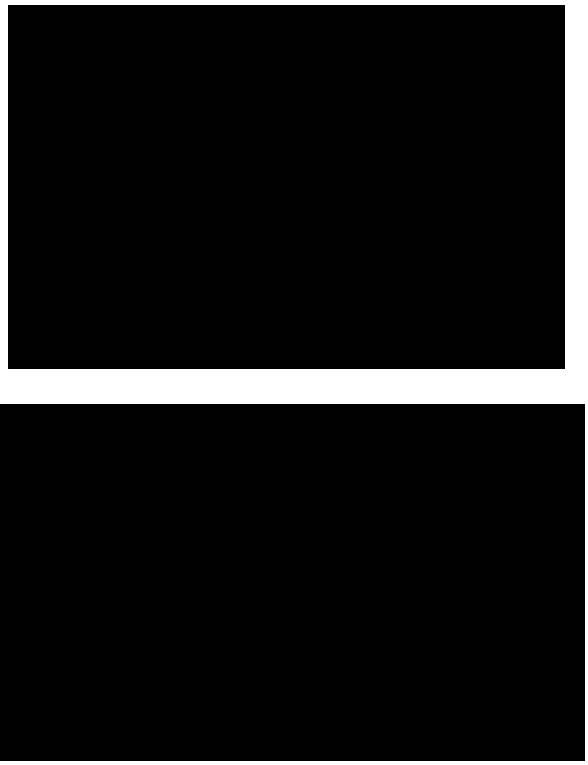


Figure 9: Makespan and Average Wait-Time versus Number of Time Slots.

Future Work

Several potential extensions to the current work can be explored. An exciting topic concerns the use of process migration, the second regards implicit scheduling, and third on the use of statistics based on real workload. The scalability of the scheduling algorithms needs to be closely looked into as well.

Migration appears to improve performance of the Gang scheduling[15]. It 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 requesting large number of nodes.

The workload was randomly generated in this study. A real application could exhibit different result and hence impact the outcomes. We need to look at this situation as well and run several benchmark processes to measure the outcome.

Concluding Remarks

In this paper we reported how a complex task such as scheduling in multiprocessor environment could be broken in manageable pieces and assigned to several students as term projects. Our experiments suggest that students appreciate working with such challenging projects in the High Performance Computing field. As pilot study, we targeted an advanced operating systems course which had sufficient components for students to work with a HPC project.

Our obtained results from the scheduling case study suggest that this project can be extended even further to include more sophisticated scenarios such as process migration, load-balancing of processor allocation, use of real tasks in the simulated scheduler instead of randomly generated payload, and scaling the number of processors. Regarding the topic of scheduling, more studies are needed to analyze behavior of the backfilling schemes as well as Gang scheduling.

Bibliography

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

2. Rajaei, Hassan and Dadfar, Mohammad, "Comparison of Backfilling Algorithms for Job Scheduling in Distributed Memory Parallel System", In Proceedings of the 2006 ASEE Annual Conference.

3. Rajaei, Hassan and Dadfar, Mohammad, "Job Scheduling in Cluster Computing: A Student Project", ASEE 2005 Annual Conference, 3620-03.

4. Lawson, Barry G., Smirni, Evgenia, "*Multiple-queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems*" Department of Computer Science, College of William and Mary Williamsburg, VA 23187-8795, USA

5. Srinivasan, S., Kettimuthu, R., Subramani, V., and Sadayappan, P., "*Characterization of backfilling strategies for parallel job scheduling*". IEEE International Conference on Parallel Processing Workshops, pages 514–519, August 2002.

6. 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.

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

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

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.

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

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

12. Yu, Philip S., Wolf, Joel L., Shachnai, Hadas, *"Look-ahead scheduling to support pause-resume for video-on-demand applications"*, Multimedia Computing and Networking 1995; Arturo A. Rodriguez, Jacek Maitan; Eds, March 1995

13. Shmueli, E. and Feitelson, Dror G., 2003. Backfilling with Look-ahead 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.

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

15. Zhang, Y., Franke, H., Moreira, J., and Sivasubramaniam, A., 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.

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 he holds an MSEE from Univ. 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.