# 2006-1737: COMPARISON OF BACKFILLING ALGORITHMS FOR JOB SCHEDULING IN DISTRIBUTED MEMORY PARALLEL SYSTEM

**Hasasn Rajaei, Bowling Green State University**

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 MSE from U. of Utah.

**Mohammad Dadfar, Bowling Green State University**

# Comparison Of Backfilling Algorithms For Job Scheduling In Distributed-Memory Parallel System

**Hassan Rajaei And Mohammad B. Dadfar**

**Department Of Computer Science**
**Bowling Green State University**
**Bowling Green, Ohio 43403**
**{Rajaei, Dadfar}@Cs.Bgsu.Edu**

## Abstract

In this paper, we compare the performance of backfilling scheduling algorithms using multiple-queue and look-ahead with the basic aggressive strategy on a multiprocessor system. Schedulers employing backfilling algorithms in distributed-memory parallel system have been found to improve system utilization and job response time by allowing smaller jobs from back of the waiting queue to execute before the larger jobs which have arrived earlier. Backfilling algorithms also overcome the problem of starvation and waste of processing resources exhibited by algorithms like shortest job first and longest job first. We have implemented the backfilling scheduling algorithms with basic aggressive, multiple-queue, and with look-ahead strategy. We compare their performances and investigate the conditions for increasing the utilization and decreasing the fragmentation of the system resources.

The look-ahead backfilling scheduling algorithm attempts to find the best packing possible given the current composition of the queue, thus maximizing the utilization at every scheduling step. It reduces the mean response time of all jobs. We use simulation to evaluate the performance of the scheduling disciplines.

## 1. Introduction

We have installed a Beowulf cluster[1, 2] with 16 computing nodes in one of our instructional labs. It provides a high performance computing environment for our courses. In our previous paper[3], we focused on a single queue of jobs and discussed three scheduling algorithms in the framework of variable partitioning: Non-FCFS, Aggressive Backfilling[4, 5], and Conservative Backfilling[5, 6, 7].

In this paper we focus on the comparison of *backfilling scheduling algorithms* using multiple-queue[4], look-ahead[8, 9], and basic aggressive strategy. Our cluster computing lab provides an excellent environment for student projects in several of our courses including Operating

Systems, Data Communication, and Distributed Programming. This paper reports the results of second phase on job scheduling studies in multiprocessor environment.

Schedulers employing backfilling algorithms in Distributed-Memory Parallel System have been found to improve system utilization and job response time by allowing smaller jobs from back of the waiting queue to execute before the larger jobs that have arrived earlier. By arranging jobs in a specific order, we reduce internal fragmentation and improve utilization of the system. Backfilling algorithms also overcome the problem of starvation and waste of processing resources exhibited by algorithm like Shortest Job First (SJF). Conservative and aggressive backfilling algorithms usually use a single queue and ignore user priority[1]. Utilization of the system resources depends on how the jobs are packaged and the order of their execution. We have implemented the backfilling scheduling algorithms using multiple-queue and dynamic algorithms using two look-ahead strategies. We compare their performances and investigate the conditions for increasing the utilization and decreasing the fragmentation of the system resources.

The multiple-queue backfilling scheduling algorithm[4] is based on aggressive backfilling strategy; it continuously monitors the system for the incoming jobs and organizes them in different waiting queues. There are four waiting queues to separate short jobs from the long ones. When new jobs arrive, the scheduler rearranges the jobs according to their estimated execution time. The system is divided into variable partitions and each partition has the same number of processors. However, if a processor is idle in one partition it can be used by the jobs in another partition. Consequently, the partition boundaries of the system are dynamic.

The look-ahead backfilling scheduling algorithm[8, 9] attempts to find an optimal configuration for the multiple queues. It tries to maximize the utilization at every scheduling step, thereby reducing the mean response time. The jobs are divided into two parts: running and waiting. The jobs that are waiting may be either in the waiting queue or in the selected queue (for execution). All jobs have two attributes: size (number of processors or computing nodes required) and estimated computing time remaining. The system free capacity is defined as the number of idle processors currently not assigned to any jobs. The main task of this algorithm is to select jobs from the waiting queue and assign available processors to them to maximize utilization.

In Sections 2 and 3 we briefly describe each of these algorithms. In Section 4 we discuss the implementation issues. Section 5 shows the simulation results. Future work and concluding remarks appears in Sections 6 and 7 respectively.

## 2. 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 | < | $t_e$ | | |

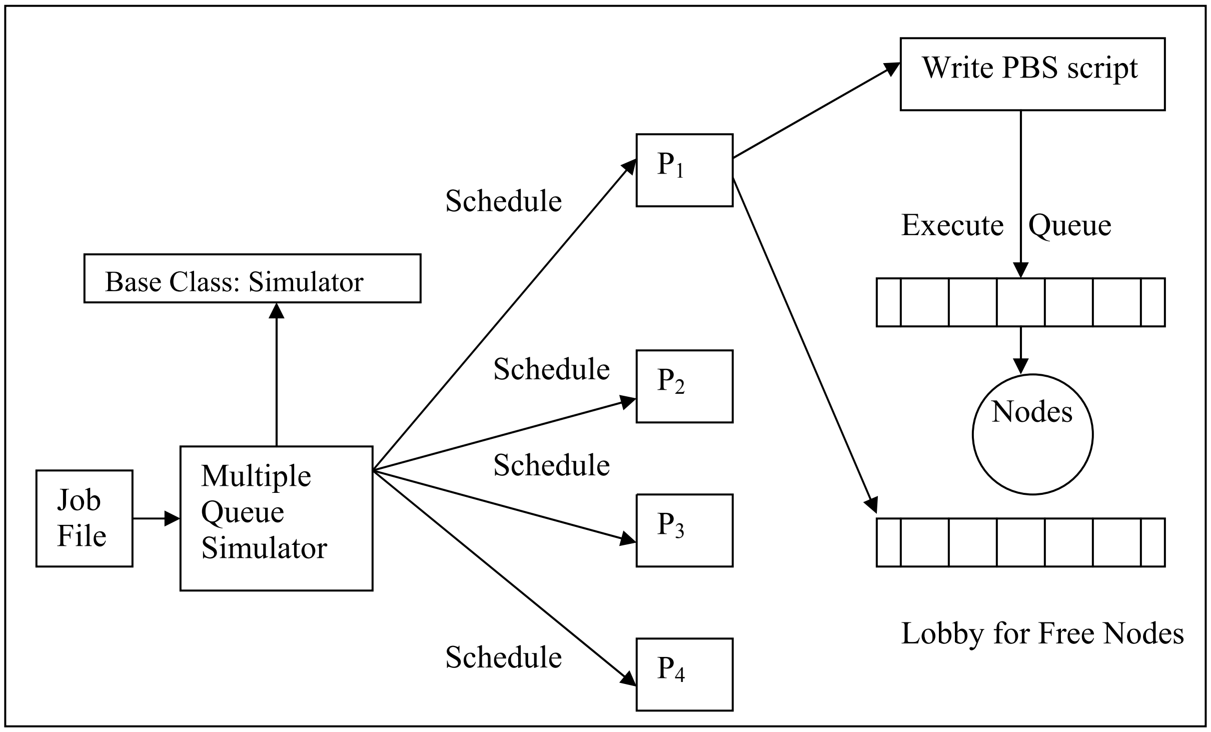

**Figure 1:** Overview of Multiple Queue Backfilling Scheduler simulator

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 1). 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 1, for example $P_1$. It holds jobs whose execution time ranges from 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 Node*s*) the scheduler searches for free nodes from other queues ($P_2$, $P_3$, $P_4$) in order 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.

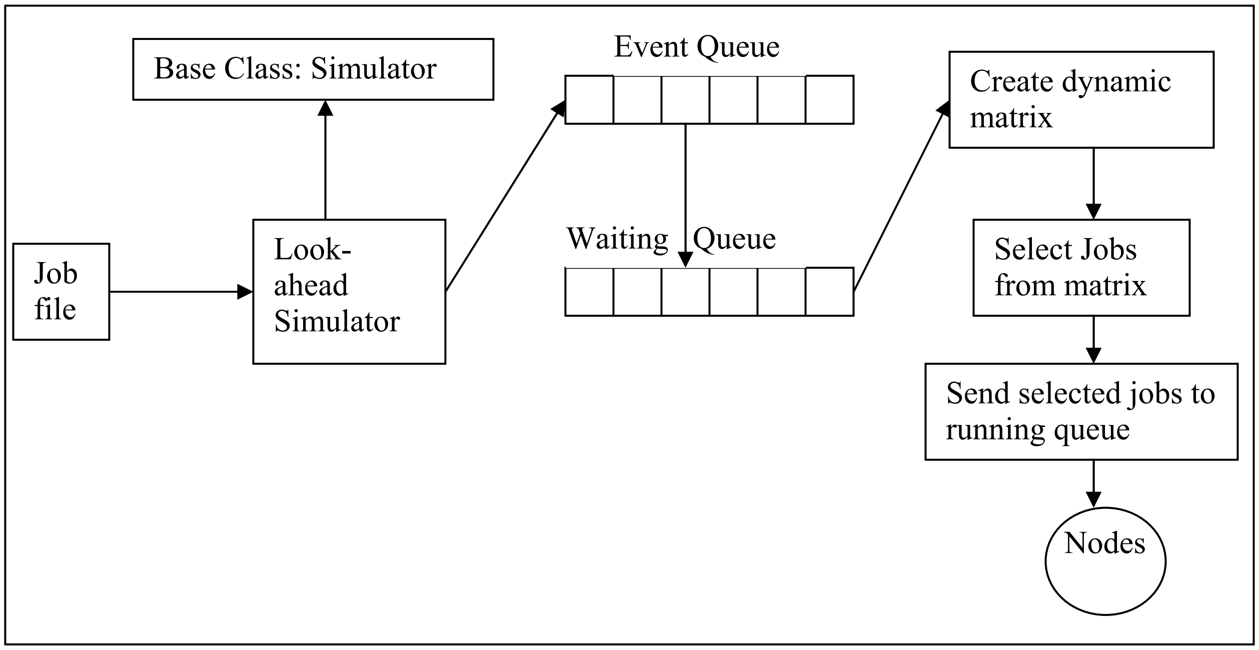

**Figure 2:** Overview of Look-ahead Backfilling Scheduler simulator

## 3. 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. 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 jobs selected for execution. All the 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.

In Figure 2, 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 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.

## 4. Implementation and Interfaces

Some code is similar for both of the scheduling algorithms. Consequently, in our implementation we use a base class called *Simulator* which contains all similar functions. From this base class, the three needed types of scheduler are derived. This method is illustrated in Figure 3 with the *Basic Aggressive*, *Multiple-Queue*, and *Look-ahead*.

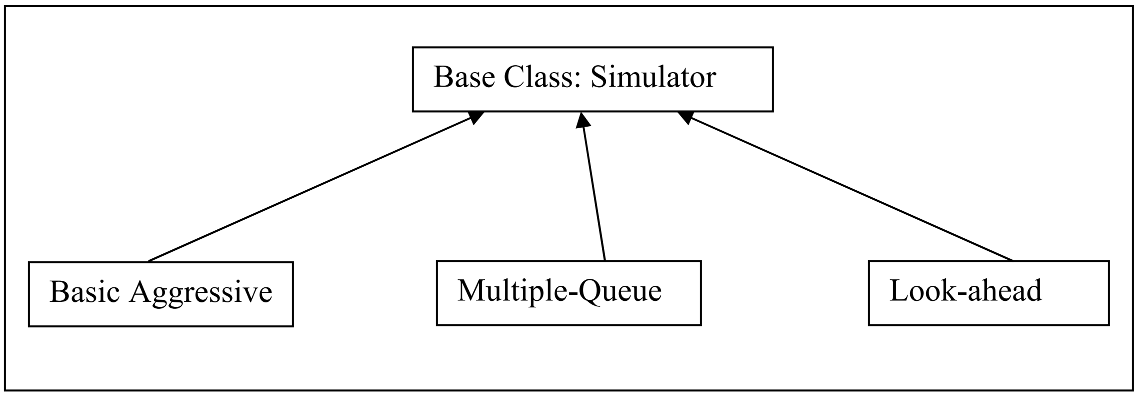

**Figure 3:** The class hierarchy of the simulators

### 4.1 Methods of Multiple Queues

Our implementation uses two methods for multiple queues: *schedule* and *run*.

a. `schedule()`

This method schedules the incoming jobs arriving at different times. Depending on the estimated execution time, the *schedule* method sends the jobs into queue $P_1$, $P_2$, $P_3$ or $P_4$ as described in Section 2. These are the waiting queues of the multiprocessor system, where the jobs are awaiting for the execution.

b. `run()`

When the scheduler determines that resources are available for the first job in a ready queue, it moves the job to the *Execute Queue*. The method *run* executes all the incoming jobs in the *Execute Queue*.

### 4.2 Methods of Look-ahead Backfilling

Following methods are used for implementation of the look-ahead backfilling algorithm:

a. `fillMatrix()`

This method creates a dynamic matrix containing the jobs that are in the *Waiting Queue*. The size of the matrix is $(|WQ|+1) \times (n+1)$. It continuously refreshes this matrix as the jobs in the *Waiting Queue* are changed.

b. `selectJobs()`

This method selects jobs from *Waiting Queue*. It checks the value of *util* and the selected value of the jobs in the matrix to select a job for execution. It stores all selected jobs in another queue.

c. `addToRQ()`

This method sends all the jobs that are in the *Select Queue* to another queue called *Running Queue* from where the jobs are executed.

d. `refreshWQ()`

When the jobs in the *Select Queue* are sent to the *Running Queue* this method refreshes the matrix by re-calculating the *util* value of the remaining jobs.

### 4.3 Common Methods

In addition, there is a common method used by all algorithms: `getNumOfFreeProc()`. The schedulers call this method to find the available number of free processors.

## 5. Simulation Examples, Results and Analysis

**Examples**: We evaluated the performance of three algorithms; basic aggressive backfilling algorithm, multiple queue backfilling, and look-ahead backfilling algorithm. The system had 30 jobs which were scheduled in the order shown in Table 1.

**Results and Analysis**

The results are show in Figure 4. Note that the waiting time of the jobs in the Multiple Queue is more than the waiting time in the Look-ahead backfilling algorithms. The waiting time of the jobs in the basic aggressive is very small as compared to the other two.

**Table 1:** Job scheduling order to the system for the three algorithms

| Job ID | Arrival Time | Estimated Time | Nodes Requested |
|---|---|---|---|
| 1 | 4 | 10 | 1 |
| 2 | 5 | 20 | 1 |
| 3 | 2 | 50 | 3 |
| 4 | 1 | 54 | 2 |
| 5 | 8 | 50 | 4 |
| 6 | 7 | 86 | 2 |
| 7 | 10 | 71 | 3 |
| 8 | 12 | 82 | 4 |
| 9 | 11 | 91 | 2 |
| 10 | 4 | 215 | 2 |
| 11 | 2 | 210 | 1 |
| 12 | 1 | 220 | 1 |
| 13 | 6 | 250 | 3 |
| 14 | 10 | 254 | 2 |
| 15 | 11 | 250 | 4 |
| 16 | 12 | 286 | 2 |
| 17 | 15 | 271 | 3 |
| 18 | 16 | 282 | 4 |
| 19 | 4 | 291 | 2 |
| 20 | 1 | 215 | 2 |
| 21 | 6 | 1310 | 1 |
| 22 | 5 | 1320 | 1 |
| 23 | 8 | 1350 | 3 |
| 24 | 9 | 1354 | 2 |
| 25 | 10 | 1350 | 4 |
| 26 | 12 | 1386 | 2 |
| 27 | 14 | 1371 | 3 |
| 28 | 7 | 1382 | 4 |
| 29 | 4 | 1391 | 2 |
| 30 | 5 | 1391 | 2 |

If we look at the line graph of each algorithm (Figure 4) separately, it seems that all three algorithms have one thing in common; the execution jobs do not depend on the arrival time. The jobs arriving late may execute before the other jobs that arrive before them. Thus, the algorithms are not fair and do not preserve the First Come First Serve principle. In case of look-ahead, the waiting time depends upon the utilization value of the job at that particular instant of 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.

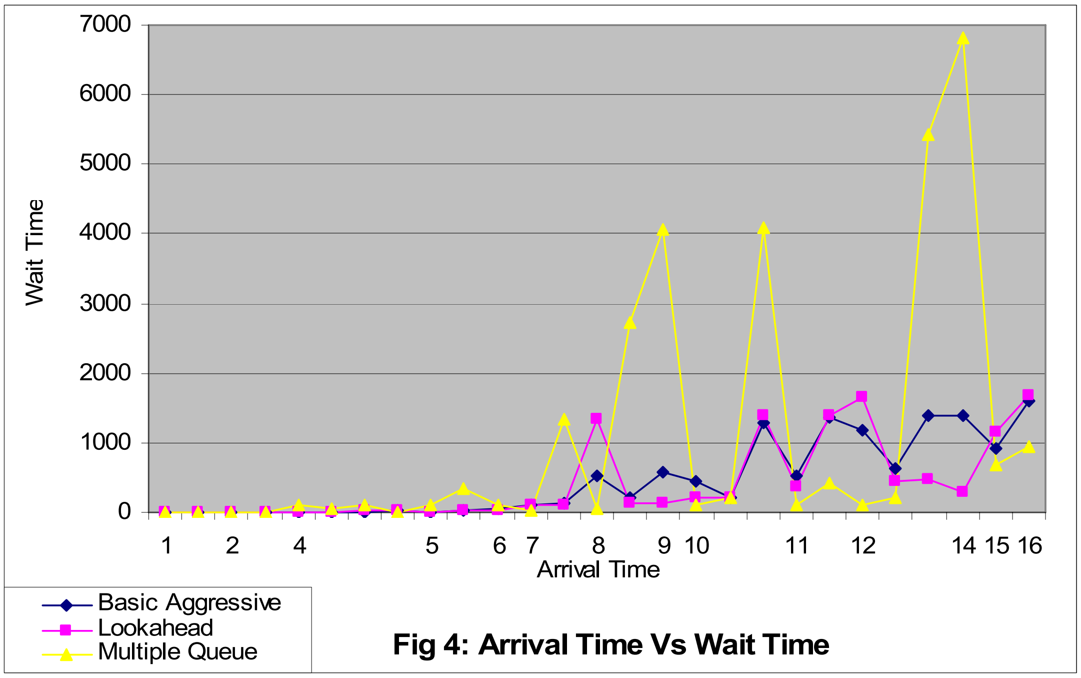

**Figure 4:** Arrival Time versus Wait Time of the three algorithms

## Requested Nodes versus Waiting Time

Figure 5 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 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

Figure 6 shows the waiting time of jobs based on their estimated time of execution. Normally, jobs with shorter estimated time are executed before jobs with larger estimated times. However, our result suggests 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.

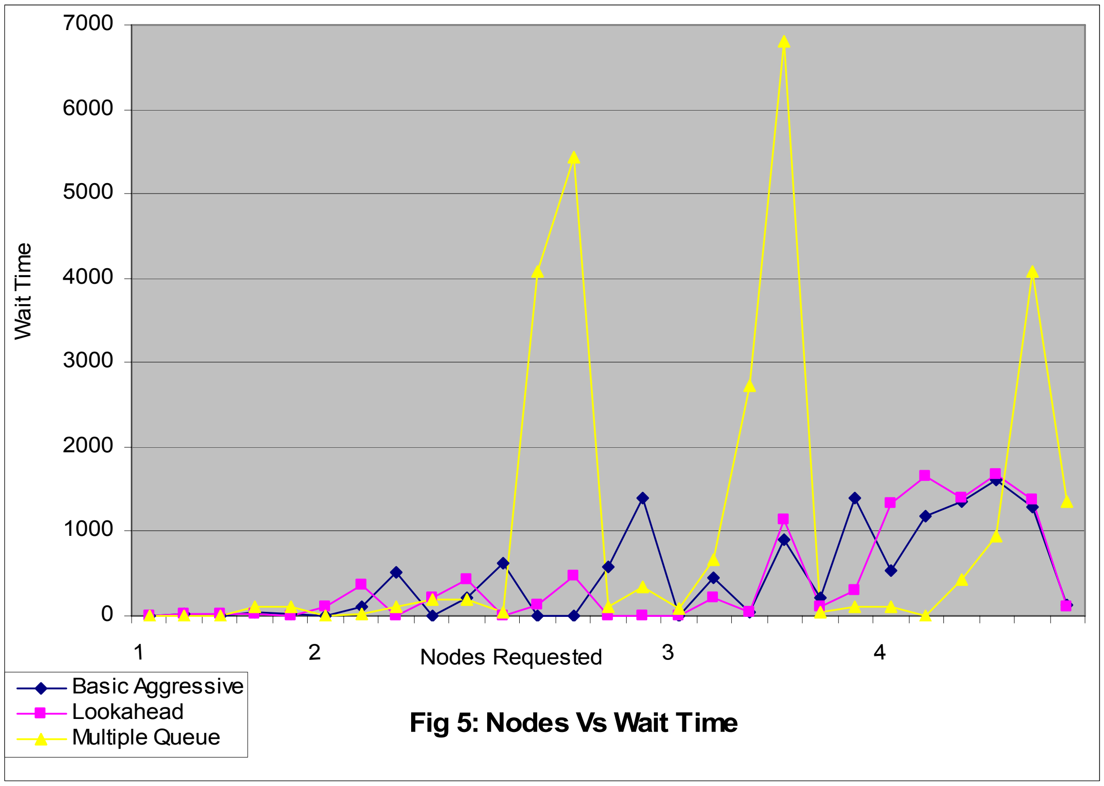

**Figure 5:** Nodes Requested versus Waiting Time

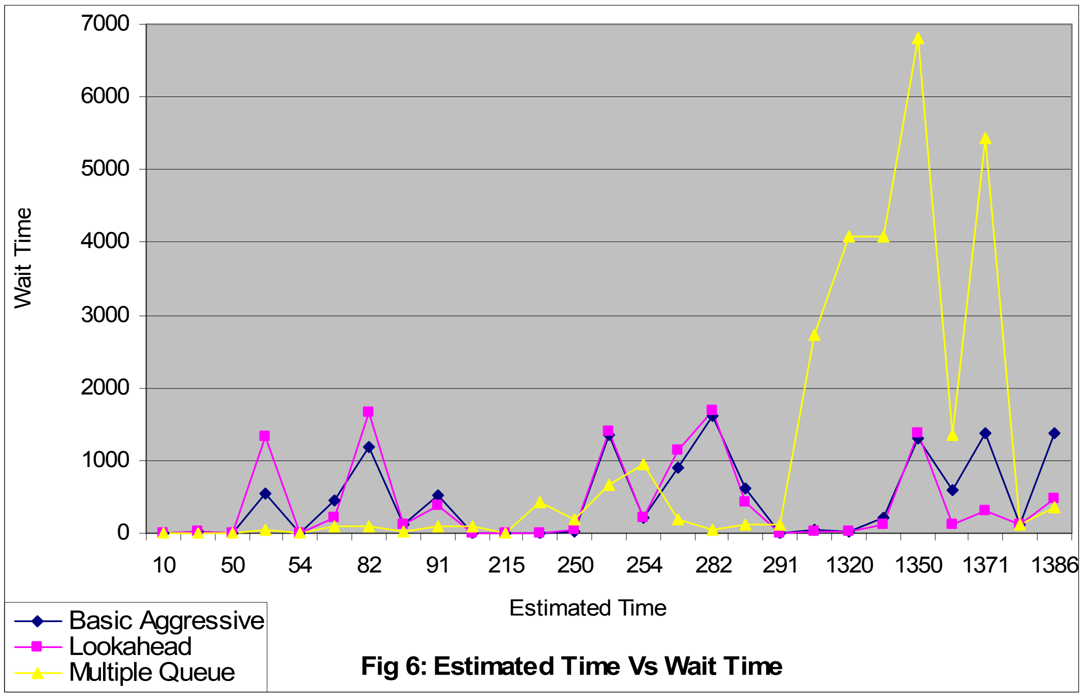

**Figure 6:** Estimated Time versus Waiting Time

## 6. Future Work

There are several potential extensions to this work. The problem of starvation in the basic look-ahead scheduling algorithm needs closer examination. The algorithm creates a subset of selected jobs from the matrix. The selected job is based on the number of requested computing nodes and nodes currently available. If a job requires more nodes, there is a possibility that the job might starve. Consequently, there should be an aging mechanism to track how long a job waits and how to make all the nodes available for those jobs that require a large number of nodes[8].

A second avenue is to explore gang scheduling and co-scheduling. In case of gang scheduling, the tasks can be grouped into a gang and concurrently scheduled on distinct processors[11, 12]. Each gang may execute in different time slot as a time sharing system. This is in contrast to batch scheduling algorithms which are non-preemptive. In case of *co-scheduling*, a job does not execute until it receives a special message from the master node.

## 7. Concluding Remarks

Look-ahead algorithm is simple and does not divide the system into variable partitions. The algorithm proposes an easy way to reschedule the incoming jobs according to their utilization value at that time. Our results suggest that look-ahead scheduling performs better than the other two. In general this algorithm has better performance when the number of processors requested by a job is small.

In Multiple-Queue algorithm, when the jobs require more processors than available in their partitions, the scheduler needs to check for the free processors. Performance improves as more processors become available. In our experiments, we have not tested this algorithm on clusters having more than 16 processors. However, we predict that it can perform better than look-ahead for larger clusters.

## Acknowledgement

We would like to thank Ms. Kavitha Alagusundaram, Ms. Roshita Mukhia, and Mr. Pankaj Joshi for their excellent work and commitment. We would like to thank Dr. Raymon Kresman for his invaluable comments which improved the quality of this paper.

## Bibliography

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

**2.** 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.

**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.** 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.

**8.** Edi Shmueli, Edi, Feitelson, Dror G., "*Backfilling with Look-ahead to Optimize the Packing of Parallel Jobs*", Department of Computer Science, Haifa University, IBM Haifa Research Lab and, School of Computer Science & Engineering, Hebrew University, Jerusalem respectively, Israel

**9.** 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 http://bookstore.spie.org/index.cfm?fuseaction=DetailPaper&ProductId=206081&coden=

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

**11.** 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.

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

**HASSAN RAJAEI**
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**
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.