Beowulf Mini-grid Scheduling

Analytical Modeling, Metascheduling, Beosim, My Thoughts

William M. Jones, Louis W. Pang

June 2003

This document has been deprecated.
Please refer to our updated
Multi-cluster Parallel Job Scheduling Website:
http://www.parl.clemson.edu/beosim

Abstract

This document contains a relatively unorganized collection of information related to my PhD research here at Clemson University. It briefly describes Beosim, the Beowulf Simulator, the analytical queueing model used to verify its statistical measurements, meta-scheduling algorithms, emperical results obtained from Beosim, and general background research related to cluster scheduling and resource management.

Contents

1 Introduction to Beosim . . . **3**

2 $M/M/m$ Model . . . **6**
- 2.1 Simple verification . . . 8
- 2.2 Additional verification . . . 8

3 Background Research . . . **8**
- 3.1 Scheduling – A Survey . . . 8
  - 3.1.1 Criteria . . . 8
  - 3.1.2 First-Come-First-Serve . . . 9
  - 3.1.3 Shortest-Job-First . . . 9
  - 3.1.4 Priority . . . 9
  - 3.1.5 Round-Robin . . . 10

3.1.6 Multilevel Queue . . . 10
3.1.7 Multilevel Feedback Queue . . . 10
3.2 IEEE TPDS, SA's [2] . . . 10
3.3 IEEE TPDS, Adaptive Computing on the Grid using AppLes [3] . . . 11
3.4 IEEE TPDS, Backfill, coscheduling, and migration [17] . . . 11
3.5 IPDPS2003: Scheduling Papers . . . 12
3.5.1 IPDPS2003: Max. Util. Co-allocation in Multiclustersystems . . . 12

4 Implemented Metascheduling Algorithms In Beosim **14**
4.1 FCFS . . . 14
4.2 FCFS_SCAN . . . 14
4.3 SMJF . . . 14
4.4 SJF . . . 14
4.5 SJF_SCAN . . . 14
4.6 SWJF . . . 15
4.7 SWJF_SCAN . . . 15
4.8 Priority . . . 15
4.9 Priority_SCAN . . . 15
4.10 FCFS_SCANBackfill . . . 15
4.11 FCFS_SCANBackfill_Var . . . 15
4.12 FCFS_SCANNoStarvation . . . 16

5 To-Do List **17**
5.1 Capabilities . . . 17

6 Some Results **18**

7 To-Do List **23**
7.1 NLN Allocation . . . 23
7.2 Moldable Jobs . . . 23
7.3 Advance reservation and Backfill . . . 24
7.4 Meta-scheduling Ideas . . . 24

List of Figures

1 Basic system . . . 4
2 Beosim Callgraph . . . 4
3 Queueing system viewed as disjoint clusters of nodes . . . 5
4 Queueing system viewed as as processor pool . . . 6
5 System Model . . . 6
6 Utilization . . . 19
7 Queue Waiting Time . . . 20
8 Utilization . . . 21
9 Queue Waiting Time . . . 22
10 Utilization . . . 23

11 Queue Waiting Time . . . 24
12 Utilization . . . 25
13 Queue Waiting Time . . . 26
14 Queue Waiting Time . . . 28
15 Cluster Utilization . . . 29
16 Queue Depth . . . 30
17 Queue Waiting Time . . . 31
18 Cluster Utilization . . . 32
19 Queue Depth . . . 33

List of Tables

1 Introduction to Beosim

The simulator itself models the behavior of a computational mini-grid by simulating incoming jobs arriving to queues within each cluster that comprises the grid. Currently jobs arrive to the queues according to an arrival process described by random variables. A scheduler determines the ordering of the queues and which jobs will be executed when resources become free. An allocator determines a mapping between a given job and the available resources, possibly including nodes from both the cluster associated with the respective queue as well as nodes in other clusters, depending on resource requirements of the job and global node availability. A dispatcher then dispatches the jobs as nodes become free in the clusters.

Each job is described by the following attributes:

1. number of nodes it requires
2. amount of time that it will run on those nodes
3. its local priority
4. its arrival time

The runtime of a given job is determined by the use of random variables as well as well as other potential factors.

The basic system can be seen by Figure 1.

Additionally, the function call graph for the current implementation is as seen the Figure 2.

In order to obtain an analytical model for such quantities as the average depth of the queues, waiting time, etc, it is necessary to simplify the nature of the system to make the analysis tractable. For example, in the current system, it can be the case that when a job is finished and departs the system, the amount of resources that are freed may not be sufficient for *any* jobs that are in the queues, not to mention the jobs that are at the head of the queues. Since this poses a challenging problem for the mathematical analysis, we will assume that when a departure occurs, if there are any jobs waiting in the queues, exactly one job will be able to make use of the freed

Figure 1: Basic system

Figure 2: Beosim Callgraph

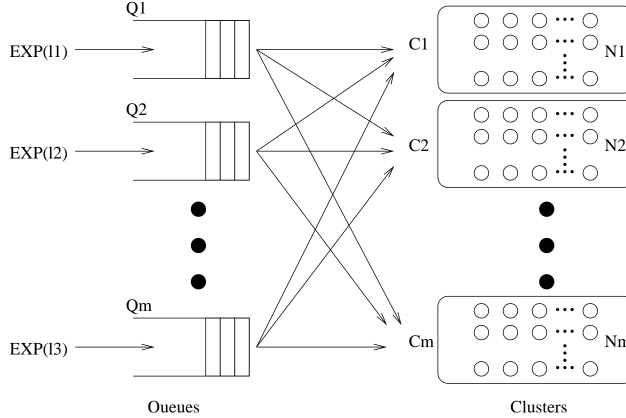

Figure 3: Queueing system viewed as disjoint clusters of nodes

resources. This is accomplised by requireing all jobs to be the same size, as well as having the total number of resources be an integer multiple of this chosen job size.

Additionally, our scheduler currently is capable of reordering the queues so that, for example, the shortest job will be at the head of its queue. Since priority is being given to the jobs based on their size, this would introduce additional complexity into the model. For now, it will be assumed that all jobs retain their FCFS ordering within a given queue.

The meta-scheduler also can inspect the jobs in each of the local queues so that it can choose which queue to select the next job from. Currently we are using a fixed known priority between queues. Since this would also introduce complexity into the analysis, this can be changed as well. For example, we could simply perform a "round-robin" job selection between the queues, or we could randomly choose the queue to take the next job from. If these types of assumptions make the analysis tractable, then we will do this as well.

Additionally the random variable we use to describe the arrival process is currently a uniform RV. This can also be relaxed to be a exponential RV in order to make the inter-arrival times be IID EXP() random variable, thus providing a Poisson arrival process. Additionally the length of the jobs are currently UNIF() random variable, but can be constrained to be EXP() as well.

Our model can also make the length of the job depend on the processing speed of the slowest processor onto which it is mapped together with a random perturbation to add variability.. Since the nodes of the clusters can have heterogeneous processing speeds, this might also prove to be too complex for analytical analysis, and can be made to be homogeneous if need be.

Finally, the length of the job can also depend on a factor that is generated based on the fact that a given job might be mapped onto processors that are not all in the same switched cluster. The runtime would then take a hit on its runtime due to this very fact. This would most likely also prove to be intractable to model, so instead of the node resources being viewed as disjoint sets of nodes as is Figure 3, we can view the collection of processors as a single "processor pool", as in Figure 4.

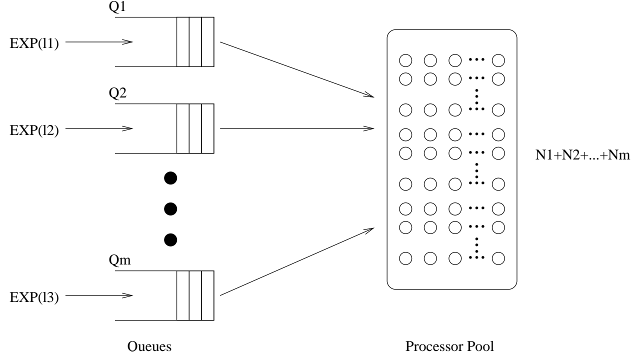

Figure 4: Queueing system viewed as as processor pool

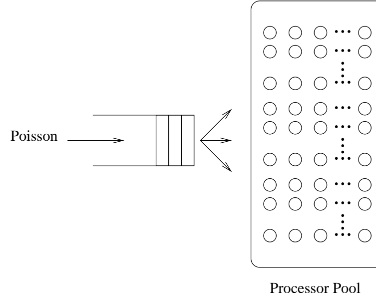

Figure 5: System Model

2 $M/M/m$ Model

It is my hope that the system can be modeled as a network of queues having a product form solution, and can be solved using techniques of MVA. I need to determine if the system has been sufficiently constrained to allow for this approach, and perhaps if some of these constraints can be relaxed to provide realism.

The model is not intended to solve the same problem as the event driven simulator does in the general case, but rather to show that the simulator is, in fact, providing reasonable results even in the most simplified of cases. As a first pass, an $M/M/m$ queueing model (Figure 5) can be used to verify the statistics provided by Beosim. This model is characterized by a Poisson arrival process with rate λ, exponential service with rate μ, a single FCFS queue, and a parallel server pool of m processors. In order to maintain the work conservation property, the assumptions are that the number of processors, n used by every job are the same and that $(m \bmod n) = 0$. Since the service rate is proportional to the number of servers currently processing jobs, it follows that $\lambda_k = \lambda, k \geq 0$ and $\mu_k = k\mu$, $1 \leq k < m$, and that $\mu_k = m\mu, k \geq m$. Since this is a birth-death process, the state probabilities, $p_k's$, are defined by

$$p_0 = \left[1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}\right]^{-1}, p_k = p_0(\lambda/\mu)^k. \tag{1}$$

By substituting for the value of λ and μ, we have that

$$p_k = \begin{cases} p_0 \frac{(\lambda/\mu)^k}{k!}, & k < m \\ p_0 \frac{(\lambda/\mu)^k}{m! m^{k-m}}, & k \geq m \end{cases} \tag{2}$$

where

$$p_0 = \left[1 + \sum_{k=1}^{m-1} \frac{\left(\frac{\lambda}{\mu}\right)^k}{k!} + \frac{\left(\frac{\lambda}{\mu}\right)^m}{m! \left(1 - \frac{\lambda}{m\mu}\right)} \right]^{-1}. \tag{3}$$

This implies that the probability mass function for the number of jobs in the queue can be found to be

$$q_k = \begin{cases} q_0 & = & q_0, & k = 0 \\ p_{k+m} & = & p_0 \frac{\left(\frac{\lambda}{\mu}\right)^m}{m!} \left(\frac{\lambda}{m\mu}\right)^k, & k > 0 \end{cases} \tag{4}$$

where

$$q_0 = 1 - \sum_{k=1}^{\infty} q_k = 1 - \frac{p_0 \left(\frac{\lambda}{\mu}\right)^m \frac{\lambda}{m\mu}}{m! \left(1 - \frac{\lambda}{m\mu}\right)}. \tag{5}$$

The z-transformation of the mass function is

$$M_q(z) = \sum_{k=0}^{\infty} q_k z^k = q_0 + \sum_{k=1}^{\infty} q_k z^k = q_0 + \frac{p_0 \left(\frac{\lambda}{\mu}\right)^m}{m!} \frac{\frac{\lambda z}{m\mu}}{1 - \frac{\lambda z}{m\mu}}. \tag{6}$$

Based on a moment-generating function identity, we have that

$$\bar{N}_q = \left. \frac{dM_q(z)}{dz} \right|_{z=1} = \frac{p_0 \left(\frac{\lambda}{\mu}\right)^m}{m!} \frac{\frac{\lambda}{m\mu}}{\left(1 - \frac{\lambda}{m\mu}\right)^2}. \tag{7}$$

By Little's theorem we can find the average waiting time in the queue to be $\bar{W} = \bar{N}_q/\lambda$. Since the average service time is $1/\mu$, the average job delay through the system is $\bar{T} = \bar{W} + 1/\mu$. Additionally, by Little's theorem we know that the average number of jobs in the system is given by $\bar{N} = \lambda \bar{T}$. Another interesting performance metric is system utilization, $\bar{U}$, which is defined by

$$\bar{U} = \sum_{k=1}^{m-1} k p_k + m * P(queueing) = \sum_{k=1}^{m-1} k p_k + m p_0 \frac{1}{m!} \left(\frac{\lambda}{\mu}\right)^m \left(\frac{1}{1 - \frac{\lambda}{m\mu}}\right), \tag{8}$$

where the probability of queueing is based on Erlang's C formula. Here $\bar{U}$ represents the average number of busy processors and can be normalized to a number between 0 and 1 by dividing by m, if so desired.

2.1 Simple verification

By using this simple model we have been able to verify the functionality of Beosim by comparing its empirical results with the analytical model. In most cases, Beosim produced statistically significant values for $\bar{W}$, $\bar{T}$, $\bar{U}$ to within a relatively small margin of error.

The following table compares the analytical model to the Beosim output for several parameterizations of λ, μ, and m. The a subscript indicates data from the analytical model whereas the b subscript indicates data from Beosim. Beosim was run using $100,000$ jobs to generate the statistical averages in the table.

λ	μ	m	$\bar{W}_a$	$\bar{T}_a$	$\bar{U}_a$	$\bar{W}_b$	$\bar{T}_b$	$\bar{U}_b$
.004	.0067	1	225.00	375.00	.60	226.45	376.02	.60
.004	.0067	2	14.84	164.84	.30	14.99	164.57	.30
.004	.0067	3	1.54	151.54	.20	1.64	151.21	.20
.005	.0059	1	963.3	1133.3	.85	1010.5	1180.1	.852
.005	.0059	2	37.5	207.5	.425	38.14	207.7	.426
.005	.0059	3	4.80	174.8	.283	4.98	174.6	.284
.005	.0059	4	.637	170.64	.213	.657	170.24	.213

2.2 Additional verification

Additionally, we have been able to verify the results of the metascheduling (in FCFS case) to show that the scheduler is maintaining the correct statistics even in the presence of multiple clusters. This is achieved my viewing the entire grid as a homogenous set of nodes, where there is no penalty of non-local node allocation. Since poisson processes are additive, you can perform a verification similar to the ones above on multiple clusters, and jobs of size larger than 1, provided that the above assumptions still hold. This is done by scaling the number of cycles each job requires, while holding the "work" (node*cycles) constant.

3 Background Research

3.1 Scheduling – A Survey

In an attempt to compile backgroud research and related works in the area of meta scheduling, the work into traditional scheduling will prove useful. This section is therefore dedicated to this task. This information is taken from [13].

CPU scheduling is essentially the task of selecting from a queue of ready processes which process is to be allocated to the CPU next. After the process is selected, the CPU is allocated to the selected process by a *dispacther*.

3.1.1 Criteria

Since different CPU scheduling algorithms have different characteristics and effects on system behaviours, it is desireable to have a set of criteria that are to be met. Traditionally these criteria are as follows:

CPU utilization	- the percent of time the CPU is busy
Throughput	- amount of work being done per unit time
Turnaround time	- time from submission to completion
Waiting time	- time spent waiting in the ready queue
Response time	- time from submission to first response

Typically the objective is to maximize the CPU utilization and throughput metrics, while minimizing the turnaround, waiting and response times. Depending on the desired system behavour, the maximization or minimization may refer to the max or min, average, or variance of the above quantities.

3.1.2 First-Come-First-Serve

Generally, the FCFS scheduling algorithm is considered to be the most simple, however the average waiting time metric is often longer than the optimal minimum average waiting time for a set of processes. The degree of variation greatly depends on the variance of the CPU-bursts associated with the various processes. Since the FCFS scheduling algorithm is generally considered to be nonpreemptive, once a process is allocated to the CPU it only releases its ownership when it terminates or requests I/O. In time-sharing systems, FCFS algorithms are particularily troublesome since it is impossible to gaurentee that each user will have access to the CPU at regular intervals.

3.1.3 Shortest-Job-First

As the name implies, the SJF scheduling algorithm selects the process with the smallest CPU-burst to go next. The SJF algorithm is provably optimal, in that it minimizes the overall average waiting time for a given set of processes. The primary issue with SJF algorithms is that you do not know apriori what the length of the next CPU-burst will be. In the case of a long-term scheduler in a batch system, the length of the process could be given by the estimate provided by the user at the time of submission. Since shorter jobs are allowed to go first, this encourages the user to accurately estimate the time needed to run the job.

In a short-term scheduler for a CPU, this is estimate is not given or known, and therefore must be predicted using some moving time average. Since this is not needed in the long-term case, it will not be discussed further.

Additionally, a SJF algorithm can either be preemptive or nonpreemptive. This distinction between the two arises when a choice has to be made when a new process arrives to the ready queue that has a shorter predicted CPU-burst.

3.1.4 Priority

In priority scheduling, each process has an associated priority. The CPU is typically allocated the process with the highest priority. In the case of processes with equal priorities, they may be scheduled according to another algorithm such as FCFS, SJF, or otherwise. Priority scheduling can either be preemptive or nonpreemptive. One of the major issues with Priority scheduling is indefinite blocking or starvation. One

solution to this problem is to allow processes to age. Generally, the effect of aging is to gradually increase the priority of a process as a function of how long it has been waiting.

3.1.5 Round-Robin

RR scheduling is essentially designed for time-sharing systems where each process is allowed a time quantum to be allocated to the CPU before it is to be switched out. This model is of switching task once they have started according to a quantum interval (even before the CPU-burst is over), is not very applicable to our endeavours. It is interesting to note however that when the quantum is very large, the RR policy is essentially the same as FCFS. In the other extreme, when the quantum is very small, the policy is equivalent to processor sharing (assuming there is not overhead associated with the process switching).

3.1.6 Multilevel Queue

With MLQ scheduling, the total set of processes are partitioned into classes according to some distinguishing characteristic such as response time requirements ranging from highly interactive *foreground* processes to batch *background* processes. Each process is assigned to one of the class partitions where each process class has its own scheduling algorithm. The process remains in the same queue for its entire lifetime.

In addition to the scheduling withing a queue, there must be scheduling between the queues. The latter is commonly implemented as a fixed-priority preemptive scheduling policy. Additionally, one could time slice between queues by allowing each queue a certain amount if CPU time.

3.1.7 Multilevel Feedback Queue

MLFQ scheduling is distinguished from MLQ scheduling in that a process can move between the queues as a function of their CPU-burst and I/O characteristics.

3.2 IEEE TPDS, SA's [2]

This article focuses on the idea of having moldable jobs, where instances of application schedulers running on a cluster can choose a node size for a job that makes more efficient use of the system, while at the same time, reducing waiting time. This contrasts some approaches in that they include fairness as a parameter in their scheduing as opposed to strictly having utilization the driving factor.

The scheduler is known as the SA, the "Supercomputer AppLeS" for Application Level Scheduler. Based on data provided about the job characteristics, the state of the supercomputer, and other information, the SA estimates the turn-around time for each request, and then suggests that the cluster execute the one with the smallest turn-around time.

The paper investigates the aggregate behaviour created by having multiple instances of the SA in the system. They also derive some performance metrics, such as *mean*

slowdown, which is like an expansion factor that forms a ratio between the turnaround time and the execution time, so that long running jobs are not over emphasized. They also introduce the idea of geometric mean of the turn-around times, in order to compare and contrast alternative scheduling solutions.

Their conecpt of modable jobs is one that can possibly have a varying number of processors at start-up time, but that cannot necessarily have a dynamic number of nodes at runtime.

One of thier cool ideas is that depending on the state of the grid, differing decisions are made in scheduling where for lightly loaded systems, the SA focuses on reducing the execution time (even if they have greate wait times) and in a heavily loaded system, the SA focuses on reducing the wait time.

One of their conclusions is that the use of multiple SA's promotes better fairness amoung all users, but that the aggregate behaviour has to be stable and not occilating in some thrashing cycle. For them this is not a problem since there is no feedback loop.

They mention future work in the areas of multi-supercomputer grid.

3.3 IEEE TPDS, Adaptive Computing on the Grid using AppLes [3]

In this paper, also by Berman, AppLes is used in a grid environment with hetrogenous resources. This paper hits on the concept that application developers and AppLes people work together to modify the original parallel code so as to work well with AppLes. The basic steps followed by an AppLes agent are as follows:

resource descovery – obvious

resource selection – selects viable resource sets, application specific

schedule generation – given feasible resource sets, generate possible schedules

schedule selection – select the best one based on some criterion

application execution – dispatch the job

schedule adaptation – iterate back to the first step, to see if system state has changed enough to warrent changing the schedule for long-running jobs

This paper is more dedicated to summarizing work that has taken place with respect to specific applications and their integration with AppLes. The paper makes some good statements in the Ongoing Work section. In particular, they are looking at AppLes templates for software reuse between AppLes enables applications, computational steering, application tunability, nondeterministic applications, global scale grids.

This paper has 97 citations.

3.4 IEEE TPDS, Backfill, coscheduling, and migration [17]

This article focuses on the idea that scheduling a job on a set of dedicated nodes over the duration of its execution time can result in low system utilization and large wait

times. They discuss three techniques that can be used beyond the traditional space sharing scheduling, namely: backfilling, gang-scheduling, and migration. They analyze the effects of combining these techniques with respect to various performance criteria.

One of their most astute observations is that since it is often the case that there are nodes that are unutilized at any given time in the cluster, eventhough there are jobs waiting in the queue, this can lead to poor utilization of the cluster. They first look at the idea of conservative backfilling, such at the holes can be filled in by taking a job out of order from the queue, and executing it.

Their second approach is to use a technique known as gang-scheduling (coscheduling). This is essentially the idea of time-shareing added to that of space shareing. They admit that this is quite effective in reducing wait time, while increasing the apparent execution time (big supprise there).

Their third approach is that of migration i.e. dynamically being able to move tasks of a parallel job. It can mitigate fragmentation in the schedule. It is important when colocation in space or time of tasks is necessary.

Workload modeling. They have an interesting synthetic workload modeling approach, by estimating the first three moments of the interarrival and service times, then fitting this to an Hyper Erlang Distribution of Common Order.

NOTE: This is a pretty good paper.

3.5 IPDPS2003: Scheduling Papers

Good papersIPDPS: [14],[6], [5], [4], [7], [15], [10], [1], [16], [8]

And TPDS: [3], [11]

And TOC: [9], [12].

3.5.1 IPDPS2003: Max. Util. Co-allocation in Multiclustersystems

In [4] Bucur and Epema address the maximal utilization of processor coallocation in multicluster systems. The idea is that since grid utilization is often much less that 100% since jobs in execution may not leave enough processors free so that any elegible job can run. They study co-allocation (mapping jobs across clusters) from a simulation as well as an analytic point of view. Their job model consists of rigid jobs that space share the grid resources, with a simple backfill policy, but they do not address time-shareing (gang-scheduling) or malleable jobs. Also, jobs a run to completion and do not change their node requirements during runtime. There are three job request types in their model:

ordered – tuple is associated with the job, where the exact number of nodes that will be required is given on a per cluster basis, and in this case, the ith index into the tuple corresponds to the number of nodes that will be required on that ith cluster, in order that the job will run.

unordered – same as ordered, except that the job does not care where which clusters the various nodes come from, just that for each index into the tuple, that number of nodes be available on a given cluster, therefore that subset of nodes cannot be

mapped across cluster boundaries. This corresponds to a job where the programmer knows apriori that the job has been decomposed into disjoint parts, where within each parition, there may be intensive communication, but that between each partion there will be little communication.

total – essentially a scalar value is provided that represents the total number of procesors that are needed for the given job, but that the job may be mapped arbitrairily across the grid without regard for how many nodes are used from which cluster.

Jobs are characterized by their sizes and their service times, which in this paper are obtained synthetically as well as from workload traces. From an analytic point of view, they are attempt to ascertain the values of such parameters as arrival rates, job sizes and service times for which the system will be stable. The arrival process is Poisson, when not otherwise know from workload files. The job component sizes (the partitions of the job) are either UNIF or something else (look at this more closely). Their distributions favor small sizes. Job service times are deterministic, exponential, hyperexponential.

They derive FCFS formulas from with p_m can be derived for the ordered requests and approximations for the unordered requests and verify with simulations. They assume tha the service times are EXP().

They introduce the term *capacity loss* to be "capacity loss l as the average fraction of the total number of processors that are idle at the maximum utilization, so $l = 1 - p_m$. They discuss 4 reasons for capacity loss. They are summarized as:

job structure – $l_{ordered} > l_{unordered} > l_{total}$

job partition size distribution – can approach pathalogical case where only one job can run at a time, and no other jobs can fit

scheduling policy – not deviating from arrival order can make l greater, but of course impilies that starvation can happen

on-line problem – decisions are made without future knowledge

These all seem relatively obvious.

They do not explicitly consider the additional communication delay introduced by the relatively slow intercluster connections (they appear to assume that all jobs are affected the same by the delay).. I need to look at this again. Also they only consider homogeneous clusters and that the only resource are computational resources.

Previously they looked at average response times and also they looked at coallocation compared to keeping all jobs local there all jobs could fit on a single cluster.

They simulate FCFS, WF placement (unordered) with service time EXP, HEXP, DET ... and they find that the the mean of the service time is immaterial, rather that the CV is the most important. They indicate the that the job component size dist has much more impact than does the service time distributions. The compare WF to FF, and FCFS to FPFS.

NOTE: I cant tell exactly to what extent backfill is used, except that when doing FPFS, the allow non-head of queue jobs to run, provided that they do not alter the

start-time of the head job, and they have a counter so that jobs can only be jumped over in the queue a certain amount of times, to prevent starvation.

4 Implemented Metascheduling Algorithms In Beosim

This section provides a listing on the various metascheduling algorithms that have been implemented so for in the Beosim simulator. Presently they all assume that the metascheduler is a central entity that has global knowledge of the local event occuring within the local clusters.

4.1 FCFS

The FCFS metascheduling algorithm globally selects the next job to run based on a strict odering in terms of the arrival time to the queue globally across the grid. This algorithm will not attempt to fill in "holes".

4.2 FCFS_SCAN

The FCFS_SCAN metascheduling algorithm globally selects the next job to run based on a strict odering in terms of the arrival time to the queue globally across the grid. This algorithm WILL attempt to fill in "holes" be sorting the queued jobs and performing a next fit with remaining available resources with respect to the arrival times.

4.3 SMJF

The SMJF metascheduling algorithm globally selects the next job to run based on running the job with the smallest number of nodes. Filling in holes is not necessary because if the smallest job can not run, neither can any other job.

4.4 SJF

The SJF metascheduling algorithm globally selects the next job to run based on running the job with the shortest runtime. It does not attempt to fill in holes.

4.5 SJF_SCAN

The SJF_SCAN metascheduling algorithm globally selects the next job to run based on running the job with the shortest runtime. It will attempt to fill in holes with the first fit with respect to the shortest job.

4.6 SWJF

The SWJF metascheduling algorithm globally selects the next job to run based on running the job with the smallest amount of work (measured in “node*cycles”). It does not attempt to fill in holes.

4.7 SWJF_SCAN

The SWJF metascheduling algorithm globally selects the next job to run based on running the job with the smallest amount of work (measured in “node*cycles”). It will attempt to fill in holes with the first fit with respect to the smallest amount of work.

4.8 Priority

The priority metascheduling algorithm globally seleects the next job to run based on running the job with the highest local priority. It does not attempt to fill in “holes”. The assumption is that global priority numbering corresponds directly to local policy schemes.

4.9 Priority_SCAN

The priority_SCAN metascheduling algorithm globally seleects the next job to run based on running the job with the highest local priority. It will attempt to fill in “holes” by performing the first fit with respect to the next highest priority. The assumption is that global priority numbering corresponds directly to local policy schemes.

4.10 FCFS_SCANBackfill

This backfill algorithm forms a global queue of arrivals to the grid, and maintains that order when considering which job to issue next. The primary difference between this version of and the normal SCAN version is that it will only allow a job to execute out of the global order provided that it will not delay the start-time of the job at the head of the queue. It might however delay the start-time of other jobs in the queue, but evenso, this algorithm prevents starvation since the queue is not reordered, eventually the job at the head of the queue will be able to run since it never gives its place up, and since it can not be delayed. This is in contrast to the aforementioned algorithms that do SCAN, since they do not preclude starvation, in particular of large jobs in the node dimention.

4.11 FCFS_SCANBackfill_Var

This backfill algorithm forms a global queue of arrivals to the grid, and maintains that order when considering which job to issue next. The primary difference between this version of backfill and the prior one is that it will allow a job to overtake the head of the queue and delay it, but only by a paramaterizable percentage of its runtime. Each time a job overtakes it that causes a delay in its start-time, the time that it is

delayed it subtracted from the total time it can be delayed, thus this figure is monotonically decreasing, and may reach 0, at which point this scheduling algorithm then behaves exactly like the FCFS_SCANBackfill algorithm. Obviously, if this algorithm is seeded with a delay percentage of 0 initially, then it behaves exactly the same as FCFS_SCANBackfill.

4.12 FCFS_SCANNoStarvation

This backfill algorithm forms a global queue of arrivals to the grid, and maintains that order when considering which job to issue next. The primary difference with this version of FCFS_SCAN is that it maintains a value associated with each job in the global queue that indicates the number of time that a job has overtaken it. A maximum number of hops can be specified, so that once a job in the queue have been overtaken that number of times, no job may overtake it. Therefore, jobs in the queue can be delayed by other jobs that are run out of order, but only a certain number of times. Since the queue is never reordered, the sequence of the number of hops associated with the jobs waiting in the queue forms a monotonically non-increasing sequence, which implies that starvation can-not occur indefinitly, but only to the extent permitted by the parameterizable MAX_HOPS.

5 To-Do List

5.1 Capabilities

As of Sep 9 2003, this is the curernt status of Beosim

Job characteristics

- Number of nodes
- Runtime

Cluster architecture

- Hetrogenous nodes
- Single queue of jobs per cluster
- Exclusive job access to a given node

Grid architecture

- Collection of of arbitrary sized clusters

Workload generation

- Completely random (EXP, UNIF, special)
- Arrival times, node req'ed, runtimes
- Tracefiles
- wkload generation on a per cluster basis

Scheduler: local

- FCFS job selection then calls metaschedhuler
- local policy on per grid basis, NOT per cluster

Scheduler: global

- FCFS, FCFS_SCAN, FCFS_SCANBackfill, FCFS_SCANBackfillVar, FCFS_SCANNoStarvation, BJF, BJF_SCAN, LJF, LJF_SCAN, Pre_Emptive, Priority, Priority_Scan, SJF, SJF_SCAN, SMJF, SWJF, SWJF_SCAN
- schedule across cluster boundaries
- currently picks job to run, not which resources to map it to
- above could be changed in current framework

Resource allocator

- creates job map for job selected by the schedulers
- or it can accept an apriori jobmap from schedulers

Logging

- logges job data that can then be ingested back into a workload tracefile
- visual information for Beoviewer, contains some instantaneous information

Summary Statistics

```
eventdriven: 30002 events processed

Final Beowulf Simulation Characteristics: (simTime = 22308324.000000)

cluster cluster1
totalJobs = 10000
Smallest Job Size = 0
Largest Job Size = 1800080
Smallest Nodes Job Size = 1
Largest Nodes Job Size = 64
Avg Nodes Req Per Job = 7.396600
Largest Waiting Time in Q = 1.4946e+06
totalRunTime = 424043833.000000
totalQueueTime = 2724993442.000000
avRunTime = 42404.4
avQueueTime = 272499
avTimeInSys = 314904
avQueueDepth = 122.151
avClusterUtilization = 0.901113
Number of nodes = 100
Throughput = 0.000448
```

Visualization: Beoview

- JAVA program
- depicts grid
- displayes some instantenous and average statistics
- ingests VISUALLOGGING file produced from Beosim

6 Some Results

After having coded some of the more simple schduling algorithms, we decided to run a few test cases with Beosim to determine how they compared. Specifically with respect to the utilization and the waiting times in the queues. The following are some of the results.

In Figures 6 and 7, the number of nodes per job is determined by a uniform random variable, UNIF(1:X), where the upper bound is varied. The job lengths are 170 (EXPRV) and the inter-arrival times are 100 (EXPRV).

In Figures 8 and 9, the number of nodes per job is determined by a uniform random variable, UNIF(X-5:X). This essentially allows a node range window size to be vaired in expected value, but constant in variance.

In Figures 10 and 11, the number of nodes per job is held constant at UNIF(1:64), whereas the inter-arrival times are varied. This also creates a change in offered load to the system, by changing the rate that jobs arrive to the system.

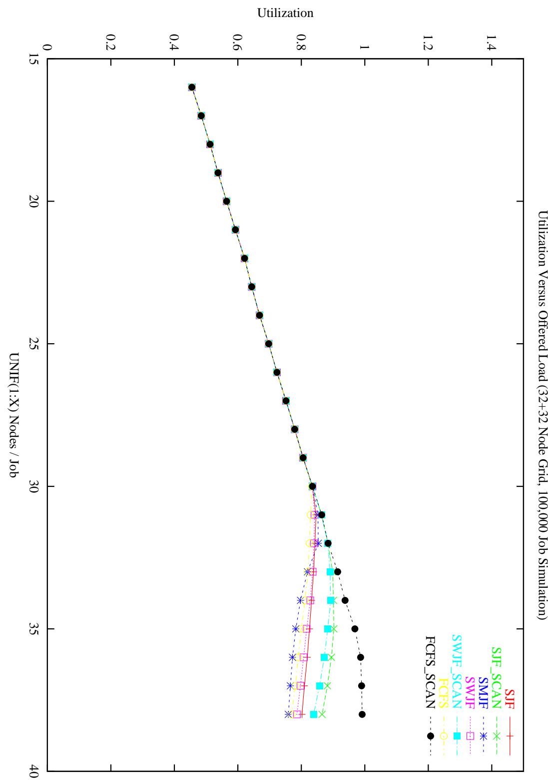

Figure 6: Utilization

In Figures 12 and 13, the number of nodes per job is determined by a uniform random variable, UNIF(X:33), where the lower bound is varied. The numCycles is 170 (EXPRV) and the interarrivalTimes is 50 (EXPRV).

On 08/06/2003, we ran the following test:

```
10000 jobs
1 cluster 100 nodes rate 1
Job size exp(7600)
resource reg unif(1,100)
interarrival(variable between 15000 -> 3000)

List of schedulers:
meta_scheduleSJFGlobal.c
meta_scheduleSJF_SCANGlobal.c
meta_scheduleSMJFGlobal.c
meta_scheduleSWJFGlobal.c
meta_scheduleSWJF_SCANGlobal.c
meta_scheduleLJFGlobal.c
meta_scheduleLJF_SCAN.c
```

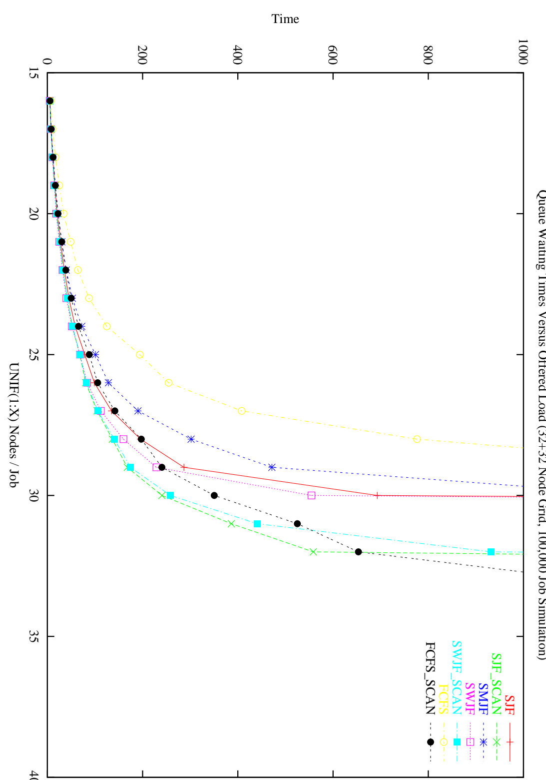

Figure 7: Queue Waiting Time

```
meta_scheduleBJFGlobal.c
meta_scheduleBJF_SCAN.c
meta_scheduleFCFSGlobal.c
meta_scheduleFCFS_SCANGlobal.c
```

During this test, we generated three output graphs 14, 15, 16:

These were the conclusions that were drawn:

For this test case we looked at some simple scheduling algorithms. We did not take into consideration backfill algorithms yet. Where possible we created a scan version counterpart to our schedulers (i.e. FCFS FCFS_SCAN). The concept of scan is given that a job can not run we scan through the queue on a FCFS basis till we find a job that can run with the current resources. This may push a job back indinitely.

For all cases we found the scan version always produced better results than the non-scan versions. This is due to the fact that we are allowing jobs to run by hopping over the current job canditate. This makes intutive sense and our test results confirms this.

We can also determine that the FCFS algorithm is not optitmal in any of the criteria we measure. Also algorithms exibit better p Srformance in one criteria while moderate performance in others. One such example is while LJF_SCAN yeilds the best

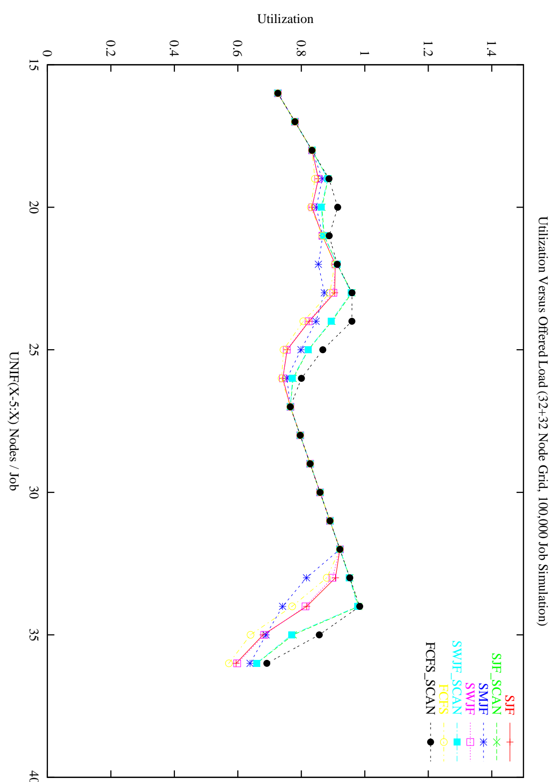

Figure 8: Utilization

utilization, it only exibits moderate performance in Qdepth and Qwaiting times.

The next test we will look at implementing backfill and starvation prevention.

```
10000 jobs
1 cluster 100 nodes rate 1
Job size exp(7600)
resource reg unif(1,100)
interarrival exp(variable between 15000 -> 3000)

List of schedulers:
meta_scheduleSJFGlobal.c
meta_scheduleSJF_SCANGlobal.c
meta_scheduleSMJFGlobal.c
meta_scheduleSWJFGlobal.c
meta_scheduleSWJF_SCANGlobal.c
meta_scheduleLJF_SCAN.c
meta_scheduleBJF_SCAN.c
meta_scheduleFCFS_SCANGlobalBackfill.c
meta_scheduleFCFSGlobal.c
```

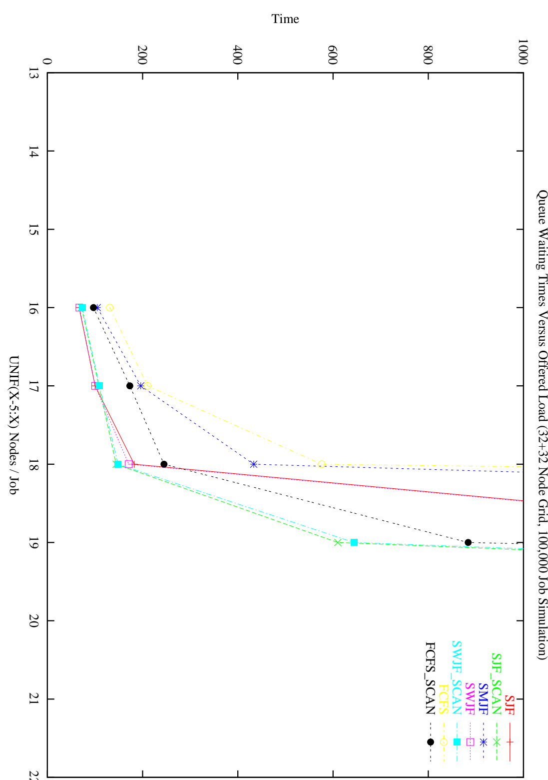

Figure 9: Queue Waiting Time

`meta_scheduleFCFS_SCANGlobal.c`

During this test, we generated three output graphs 17, 18, 19:

These were the conclusions that were drawn: For this run we eliminated LJF and BJF because neither one produced good results for any of the criteria. We added a simple backfill algorithm that backfills only the first job. Given a job canditate can not run because of limited resources we determine when it can run and scan through the queue, using FCFS, till we find a job that can run with out pushing the current job canditate further.

From the data we can see the backfill algorithm produced excellent utilition, almost 25 percent better than FCFS, and produced good queue waiting times. This backfill prevents starvation but if a job only pushes another job back 1 percent of its run time, then it will not run. For our next test we look at how the algorithm performs when the job can be pushed back some percentage of its runtime.

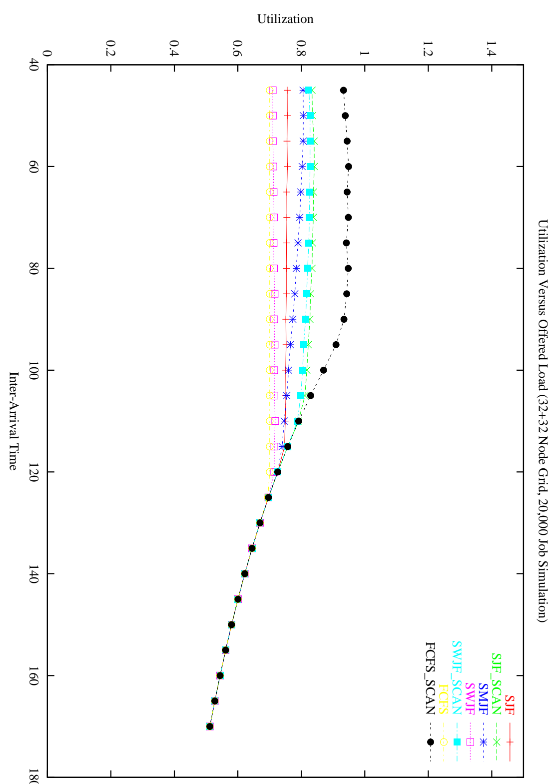

Figure 10: Utilization

7 To-Do List

7.1 NLN Allocation

A model needs to be determined for the non-local-penalty (NLP) for allocating a job across cluster boundaries. This term will most likely be a function of the fixed cost of borrowing nodes plus some other terms to be determined.

7.2 Moldable Jobs

Since there are many classes of jobs that can be run in parallel with a varying number of processors, we should modify the job model to include a range over which the job can be said to be moldable. This will need to be accompanied by a speed-up metric that will specify how much the job can be speedup. We need to look at the most recent IEEE-TPDS to look at the work done there with the simulator.

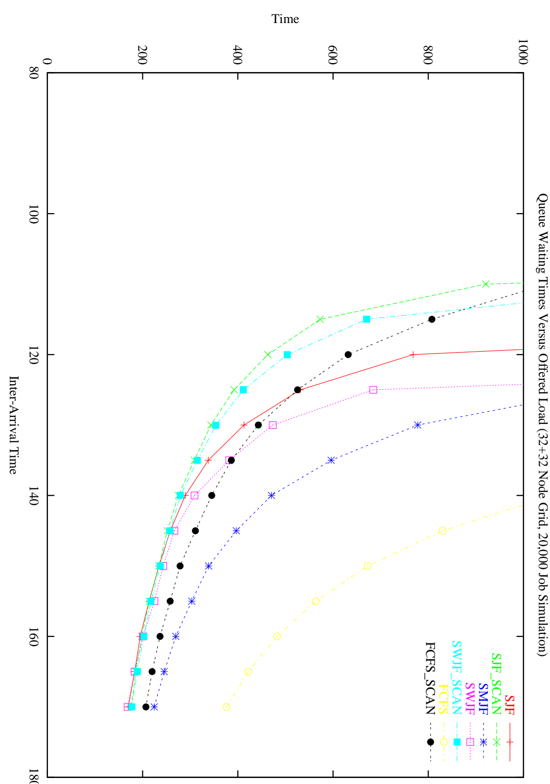

Figure 11: Queue Waiting Time

7.3 Advance reservation and Backfill

We need to implement a class of metascheduling algorithms that will provide the ability to do planning, so that we can determine under what circumstances, if any, backfill in not important.

7.4 Meta-scheduling Ideas

Originally Dr. Stanzione and I spoke of the distinction between a local scheduler residing on each cluster versus a meta-scheduler that controled the local schedulers in some global, as yet undefined, way. Of course one of my original questions was, "Why not have a global scheduler control the entire process from a centralized point, rather than complicate the matter with the additional level indirection presented by having both local schedulers as well as a meta-scheduler?" The response was that, amoung other things, as centralized control point would introduce a "single point of failure."

I now wonder if would not be possible to merge the two ideas of having a system that was simple in that a single scheduler would have access to global information, yet would not be susceptable to the inherent limitation of faults related to single point failures. It occured to me that mobile ad-hoc networks, there is often the question of

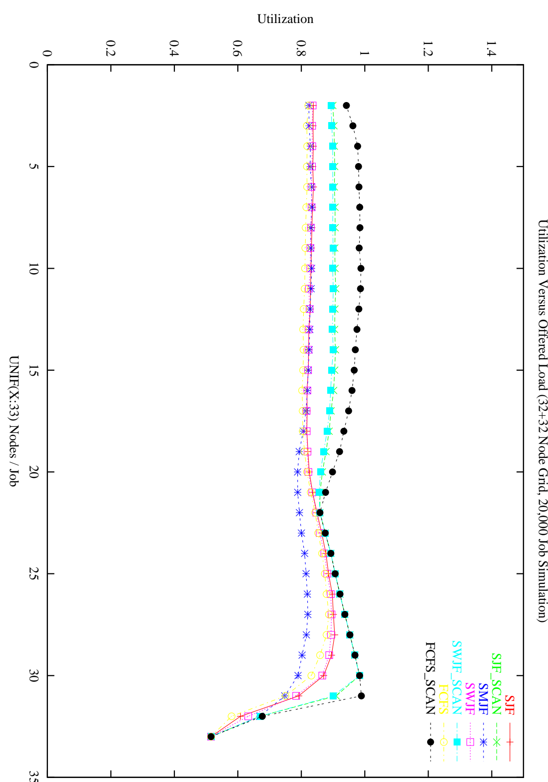

Figure 12: Utilization

a “leader election.”

Would it not be possible to have a ”potential” leader node present in each cluster that would he capable of taking on the role of a global scheduler? In the event that a fault occured at the current leader, a new leader would be elected via some leader election scheme ? Since our mini-grid would not suffer from that same transients as an ad-hoc network, it would most likely not require the frequent reassignment of leadership that mobile networks experience. Additionally, since we are not as concerned with the real-time constraints that typify other such networks, the global scheduler could have access to global information in order to make its scheduling decisions.

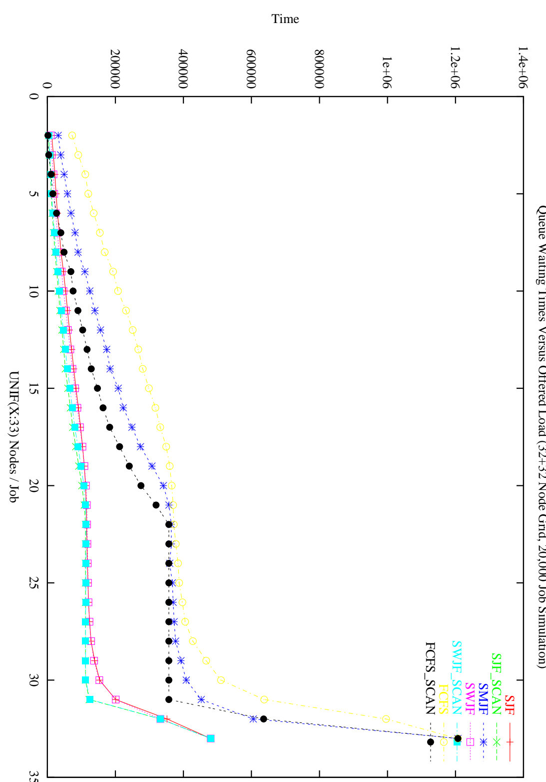

Figure 13: Queue Waiting Time

References

[1] Alexander Barmouta and Rajkumar Buyya. GridBank: A Grid Accounting Services Architecture (GASA) for Distributed System Sharing and Integration. In *International Parallel and Distributed Processing Symposium*. IPDPS2003: Workshop on Internet Computing and E-commerce, April 2003.

[2] Francine Berman and Walfredo Cirne. When the Herd Is Smart: Aggregate Behavior in the Selection of Job Request. In *IEEE Transactions On Parallel and Distributed Systems*, volume 14, pages 181–192, February 2003.

[3] Francine Berman, Richard Wolski, Henri Casanova, and Walfredo Cirne. Adaptive Computing on the Grid Using AppLes. In *IEEE Transactions On Parallel and Distributed Systems*, volume 14, pages 369–382, April 2003.

[4] A. I. D. Bucur and D. H. J. Epema. The Maximal Utilization of Processor Co-Allocation in Multicluster Systems. In *International Parallel and Distributed Processing Symposium*. IPDPS2003, April 2003.

[5] Junwei Cao, Daniel P. Spooner, Stephen A. Jarvis, Subhash Saini, and Graham R. Nudd. Agent-Based Grid Load Balancing Using Performance-Driven

Task Scheduling. In *International Parallel and Distributed Processing Symposium*. IPDPS2003, April 2003.

[6] E. Caron, F. Desprez, F. Petit, and V. Vilain. A Hierarchical Resource Reservation Algorithm for Network Enabled Servers. In *International Parallel and Distributed Processing Symposium*. IPDPS2003, April 2003.

[7] Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. In *International Parallel and Distributed Processing Symposium*. IPDPS2003, April 2003.

[8] Jong-Kook Kim, Sameer Shivle, Howard Jay Seigel, Anthony A. Maciejewski, and et. al. A Decentralized Hierarchical Scheduler for a Grid-based Clearinghouse. In *International Parallel and Distributed Processing Symposium*. IPDPS2003: Heterogeneous Computing Workshop, April 2003.

[9] Keqin Li and Yi Pan. Probabilistic Analysis of Scheduling Precedence Constrained Parallel Tasks on Multicomputers with Contiguous Processor Allocation. In *IEEE Transactions On Computers*, volume 49, pages 1021–1030, October 2000.

[10] Vincenzo Di Martino. Sub Optimal Scheduling in a Grid using Genetic Algorithms. In *International Parallel and Distributed Processing Symposium*. IPDPS2003: Workshop on Nature Inspired Distributed Computing, April 2003.

[11] Arnold L. Rosenberg. Optimal Schedules for Cycle-Stealing in a Network of Workstations with a Bag-of-Tasks Workload. In *IEEE Transactions On Parallel and Distributed Systems*, volume 13, pages 179–191, February 2002.

[12] Tsan sheng Hsu, Joseph C. Lee, Dian Rae Lopez, and William A. Royce. Task Allocation on a Network of Processors. In *IEEE Transactions On Computers*, volume 49, pages 1339–1353, December 2000.

[13] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison-Wesley, 1998.

[14] Xian-He Sun and Ming Wu. Grid Harvest Service: A System for Long-term, Application-level Task Scheduling. In *International Parallel and Distributed Processing Symposium*. IPDPS2003, April 2003.

[15] Geoffroy Vallee, Christine Morin, Jean-Yves Berthou, and Louis Rilling. A New Approach to Configurable Dynamic Scheduling in Clusters based on Single System Image Technologies. In *International Parallel and Distributed Processing Symposium*. IPDPS2003, April 2003.

[16] Percival Xavier, Bu-Sung Lee, and Wentong Cai. A Decentralized Hierarchical Scheduler for a Grid-based Clearinghouse. In *International Parallel and Distributed Processing Symposium*. IPDPS2003: Workshop on Massively Parallel Processing, April 2003.

[17] Yanyong Zhang, Hubertus Franke, Jose Moreira, and Anand Sivasubramaniam. An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration. In *IEEE Transactions On Parallel and Distributed Systems*, volume 14, pages 236–247, March 2003.

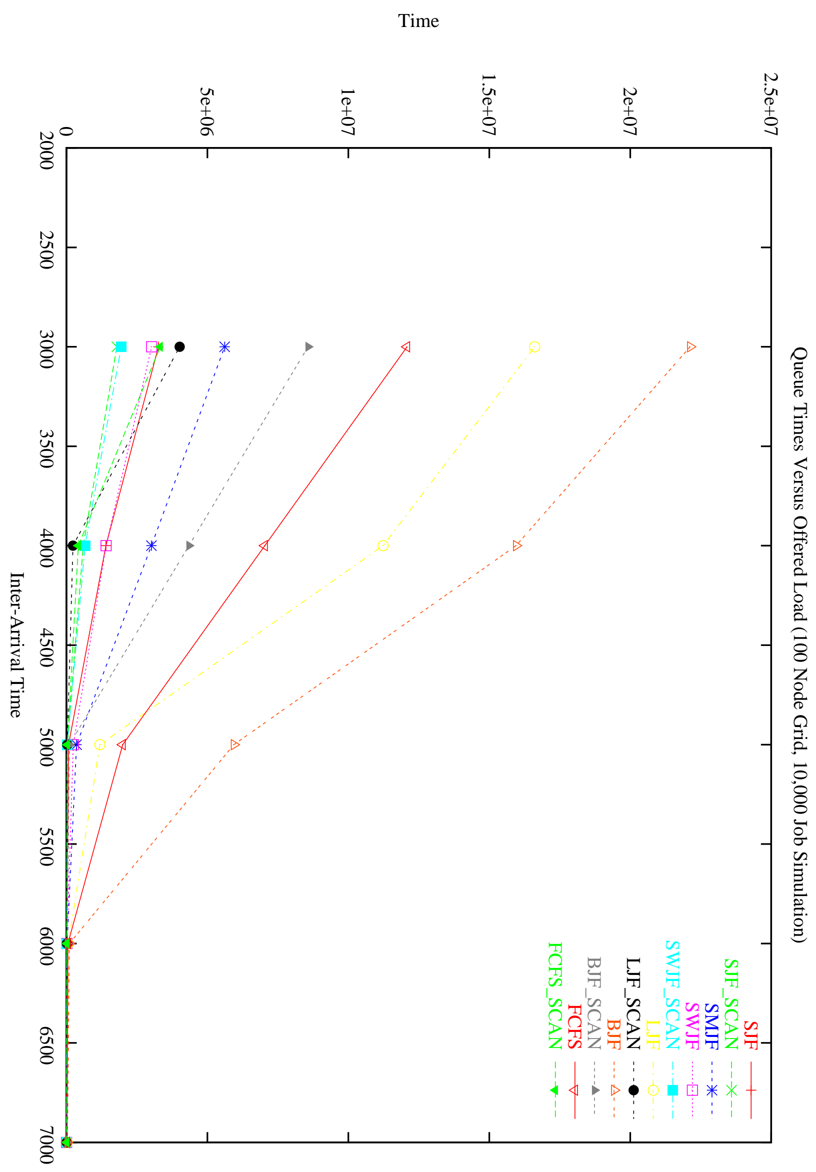

Figure 14: Queue Waiting Time

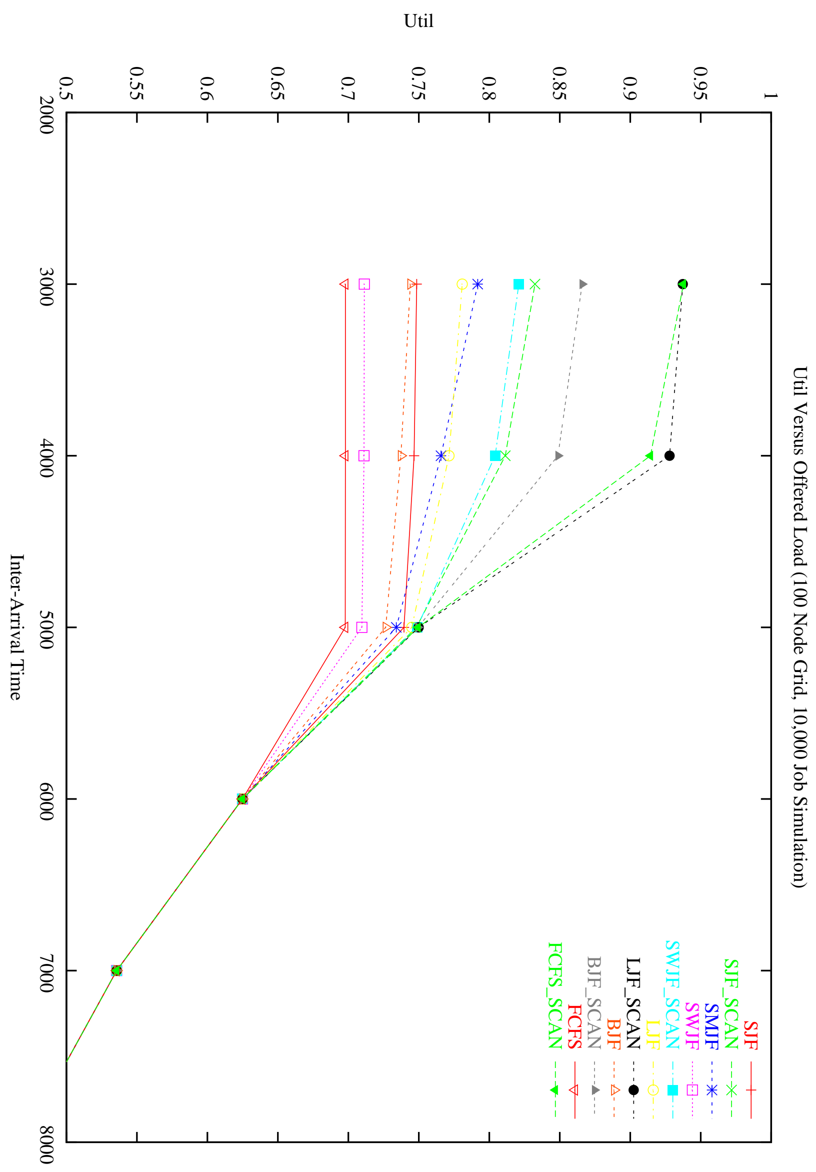

Figure 15: Cluster Utilization

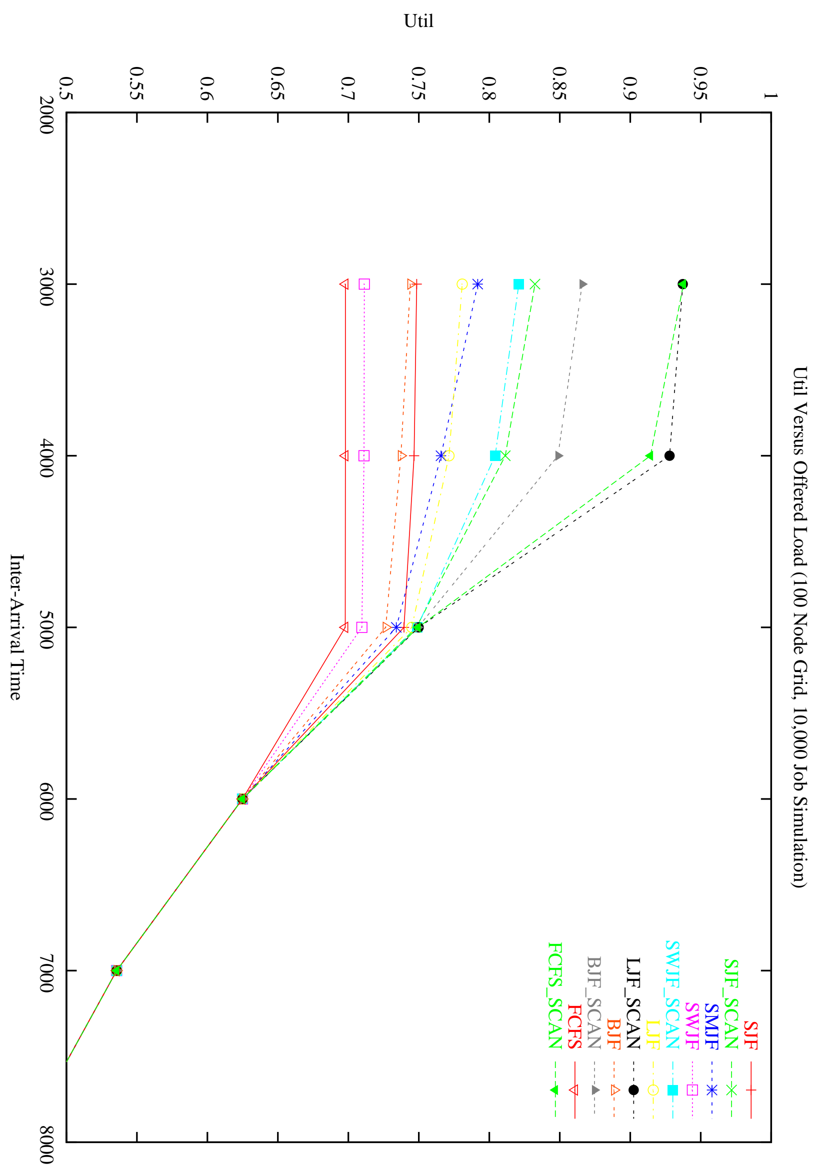

Figure 16: Queue Depth

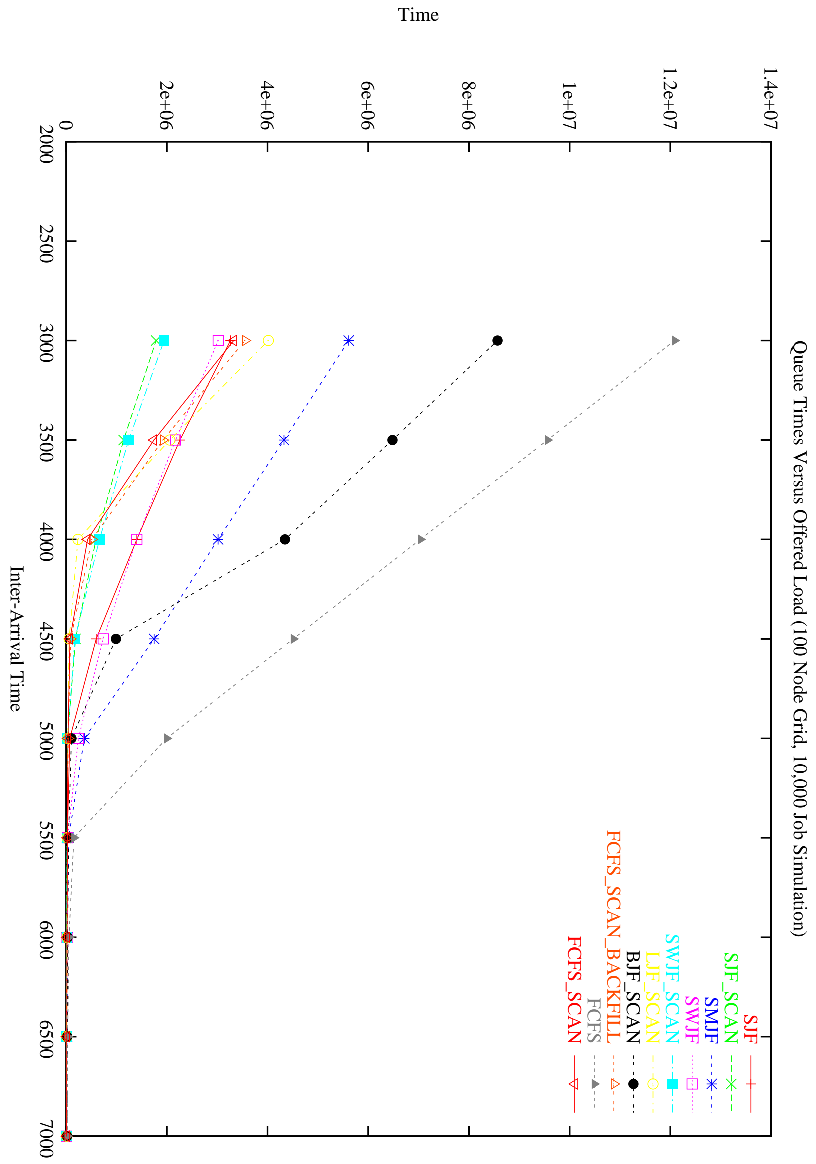

Figure 17: Queue Waiting Time

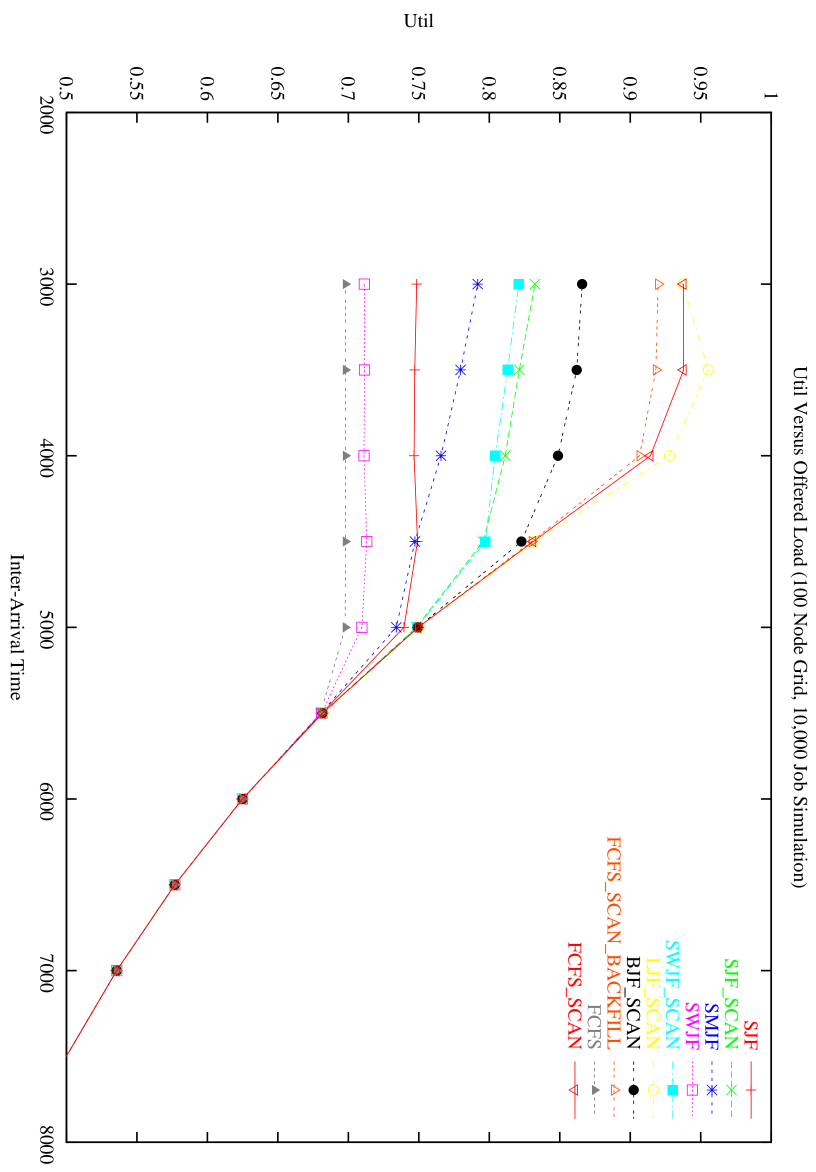

Figure 18: Cluster Utilization

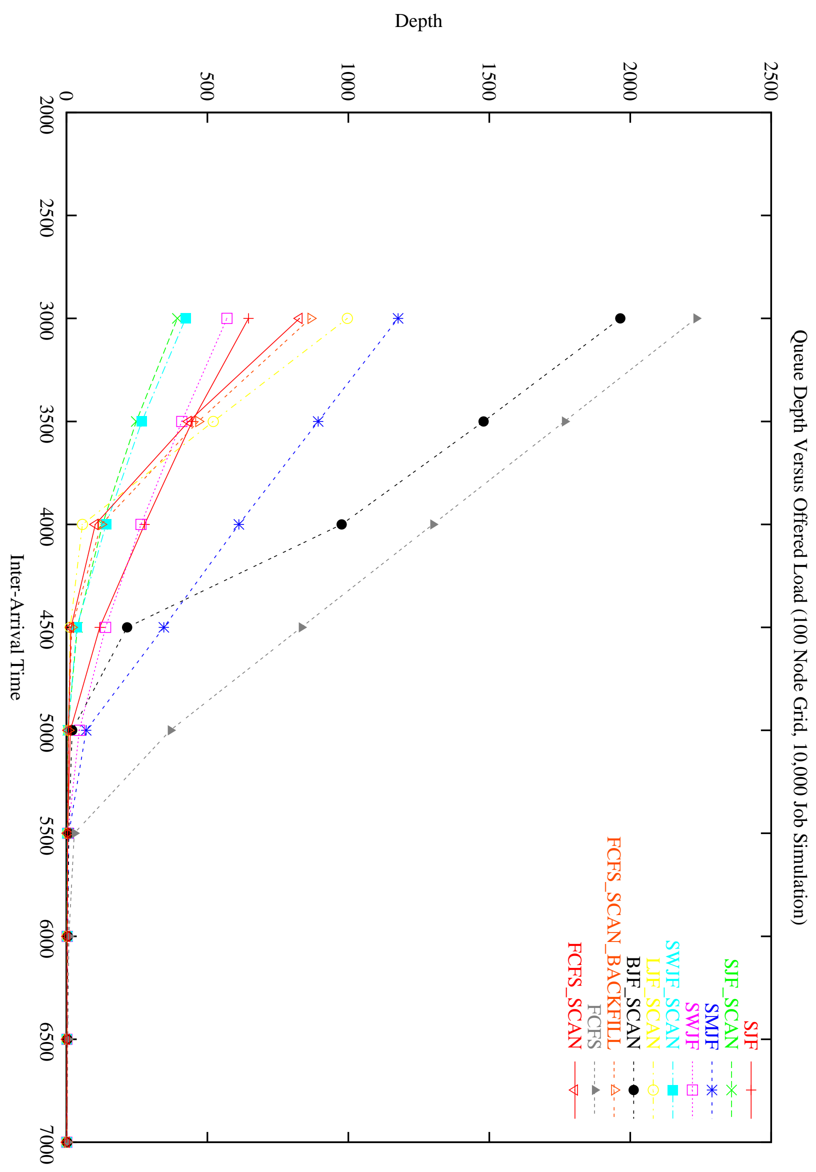

Figure 19: Queue Depth