Decentralized Preemptive Scheduling across Heterogeneous Multi-core Grid Resources

Arun Balasubramanian[1], Alan Sussman[2] and Norman Sadeh[1]

[1] Institute for Software Research, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213
arunb@cs.cmu.edu, sadeh@cs.cmu.edu
[2] Department of Computer Science, University of Maryland, College Park MD 20742
alan@cs.umd.edu

Abstract. The recent advent of multi-core computing environments increases the heterogeneity of grid resources and the complexity of managing them, making efficient load balancing challenging. In an environment where jobs are submitted regularly into a grid which is already executing several jobs, it becomes important to provide low job turn-around times and high throughput for the users. Typically, the grids employ a First Come First Serve (FCFS) method of executing the jobs in the queue which results in suboptimal turn-around times and wait times for most jobs. Hence a conventional FCFS scheduling strategy does not suffice to reduce the average wait times across all jobs. In this paper, we propose new decentralized preemptive scheduling strategies that backfill jobs locally and dynamically migrate waiting jobs across nodes to leverage residual resources, while guaranteeing (on a best effort basis) bounded turn-around and waiting times for all jobs. The methods attempt to maximize total throughput and minimize average waiting time while balancing load across available grid resources. Experimental results for both intra-node and internode scheduling via simulation show that our scheduling schemes perform considerably better than the conventional FCFS approach of a distributed or a centralized scheduler.

Keywords: Distributed systems, Scheduling, Preemptive scheduling, Performance, Load balancing, Heterogeneous processors, Grid computing

1 Introduction

Modern machines use multi-core CPUs to enable improved performance. In a multi-core environment, it has been a challenging problem to schedule multiple jobs that can run simultaneously without oversubscribing resources (including cores). Contention or shared resources can make it hard to exploit multiple computing resources efficiently and so, achieving high performance on multi-core machines without optimized software support is still difficult [15]. Moreover, grids that contain multi-core machines are becoming increasingly diverse and heterogeneous [10], so that efficient load balancing and scheduling for the overall system is becoming a very challenging problem [5][4] even with global status information and a centralized scheduler [21].

Previous research [9] on decentralized dynamic scheduling improves the performance of distributed scheduling by starting jobs capable of running immediately (backfilling), through use of residual resources on other nodes (when the job is moved) or on the same node. However, the scheduling strategy is non-preemptive and follows a first come first serve approach to schedule the jobs. This results in suboptimal wait times and turnaround times for most jobs in the queue. It also results in suboptimal overall job throughput rate in the grid.

The performance of distributed scheduling and overall job throughput in such multicore environments can be improved by following a preemptive scheduling strategy where jobs that have lower estimated running times in the queue are scheduled to run immediately. The techniques of migrating jobs to use residual resources on neighboring nodes can also be used to increase the overall CPU utilization. However, because of limited and/or stale global state information, efficient decentralized job migration can be difficult to achieve. Moreover, a job profile often has multiple resource requirements; a simple job migration mechanism considering only CPU usage cannot be applied to in such situations. In addition, guarantee of progress for all jobs is also desired, i.e., no job starvation.

The contribution of this paper is a novel dynamic preemptive scheduling scheme for multi-core grids. The scheme includes (1) local preemptive scheduling, with backfilling on a single node, (2) internode scheduling, for backfilling across multiple nodes, and (3) queue balancing, which proactively balances wait queue lengths. The approach is inspired by ideas from the preemptive schedulers in the context of operating systems, and schedules jobs at regular intervals based on its priorities. The priorities of the jobs are determined according to their remaining time for completion and the amount of time the job has spent waiting in the queue. It is a completely decentralized scheme that balances load and improves throughput when scheduling jobs with multiple constraints across a distributed system. We demonstrate the effectiveness of these algorithms via simulations that show that the decentralized preemptive scheduling approach outperforms the non-preemptive scheduler that follows a first-come-first-serve strategy.

The rest of this paper is organized as follows. Section 2 discusses the related work on various preemptive scheduling strategies in literature. Section 3 discusses the distributed scheduling strategies and describes the basic architecture of the peer-to-peer grid systems and the resource management schemes for multi-core machines. The term definitions related to the scheduling algorithm are presented in section 4. The preemptive scheduling approach is discussed in Section 5. The simulation results are presented in Section 6. Conclusions and future work are presented in Section 7 and Section 8, respectively.

2 Related Work

Various scheduling algorithms (both preemptive and non-preemptive) have been described in the literature, especially in the contexts of Operating Systems, Batch Processing and Real time scheduling environments. First-come first serve (also termed as FCFS), Round-Robin, shortest-remaining time, fixed priority preemptive scheduling are some of the scheduling algorithms that are widely in use. In classical UNIX systems [20] [2], if a higher priority process became runnable, the current process was preempted even if the process did not finish its time quantum. This resulted in higher priority processes starving low-priority ones. To avoid this, a 'usage' factor was introduced to calculate process priority. This factor allowed the kernel to vary processes priorities dynamically. When a process was not running, the kernel periodically increased its priority. When a process received some CPU time, the kernel reduced its priority. This scheme could potentially prevent the starvation of any process, since eventually the priority of any waiting process would rise high enough to be scheduled. While operating system schedulers usually act on the basis of information obtained from the processes executed so far and the priority of processes, batch processing and real time schedulers have added information, such as estimated job completion times and job deadlines, respectively. Our environment closely resembles that of the Batch Processing scenario since it is reasonable to obtain estimates of the job completion times.

Previous research [9] on distributed scheduling scheduled jobs in a FCFS fashion. Although this approach had minimal scheduling overhead, the turnaround times, waiting times and response times were high for many jobs since the long running jobs hogged the CPU. Also, no prioritization resulted in the system having trouble meeting the process deadlines. The work done by Quinn et al. [16] on preemption based backfill addresses the problem of inefficient resource utilization by backfilling lower priority jobs. The preemptive backfill technique used in the paper allows the scheduler to schedule lower priority jobs even if they cannot finish execution before the next higher priority job is scheduled. We use a similar technique for our strategies. The work on checkpoint based preemption [13] discusses employing checkpoints for preemption and improves the job scheduling performance in waiting time by addressing the inaccuracies in user-provided runtime estimates.

Shortest remaining time [6] is a scheduling method that is a preemptive version of shortest job next [18] scheduling. In this algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the executing process is the one with the shortest amount of time remaining (by definition), processes always run until they complete or a new process is added that requires a smaller amount of time. This leads to higher wait times for long running jobs. Highest Response Ratio Next (HRRN) [19] scheduling is a preemptive discipline, in which the priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting. Jobs gain higher priority the longer they wait, which prevents indefinite postponement (process starvation). i.e. the jobs that have spent a long time waiting compete against those estimated to have short run times. In this paper, we use the idea of 'Higher Response Ratio Next' in a distributed environment to ensure that long running jobs are not starved of CPU usage while at the same time guaranteeing that shorter jobs finish early. This contributes to the overall high throughput in the system.

3 BACKGROUND

Several scheduling strategies have been studied in the context of distributed computing ranging from cluster computing to the now-prevalent heterogeneous computing grids. Most of the distributed scheduling strategies in the heterogeneous environments are focused on application level scheduling [3] (i.e. they focus on how to efficiently break down and schedule the sub-tasks of the application) so as to maximize the use of the heterogeneous components like GPUs, CPUs and memory. Some research has also been done to address the issue of dynamically scheduling each incoming job by learning through past performance histories [7] and migrating jobs [9]. However, they all schedule the incoming jobs in a non-preemptive or FCFS order. Though some studies have been done(as discussed in Related Work), a comprehensive study still remains to be done on the preemptive strategies for scheduling the jobs submitted onto the grid.

Issam et al. [1] proposes a scheduling strategy which consists of policies that utilizes the solution to a linear programming problem which maximizes system capacity. This however is a centralized approach and hence has the limitations of a centralized scheduler. The paper on computational models and heuristic methods on grid scheduling by Fatos Xhafa at al. [23] exceptionally summarizes the scheduling problems involved in grid computing. It also gives good insight on the different scheduling strategies that can be used and presents heuristic methods for scheduling in grids. However, they fail to discuss in detail the benefits of the opportunities presented by a preemptive scheduling model. We then date back as early as Condor [12]; a system that employs a preemptive strategy. Although Condor does not have a preemptive centralized scheduler, the local scheduler enforces preemption of the job whenever the user resumes activity. Our scenario can be compared to this in the sense that a higher priority job (a user process in case of Condor) may be ready to run at any given instant.

A pivotal aspect to be considered before scheduling is finding the right node to run the job. Various resource discovery techniques exist in the literature that assign the incoming jobs to chosen nodes. The Classified Advertisement (ClassAd) [14] and the CAN [17] approaches are examples of distributed matchmaking algorithms that match incoming jobs to lightly loaded nodes. Matchmaking is the initial job assignment to a node that satisfies all the resource requirements of the job, and also does load balancing to find a (relatively) lightly loaded node. A good matchmaking algorithm has several desirable properties: expressiveness, load balance, parsimony, completeness, and low overhead. The matchmaking framework should be expressive enough to specify the essential resource requirements of the job as well as the capabilities of the nodes. It should balance load across nodes to maximize total throughput and to obtain the lowest job turnaround time. However, over-provisioning can decrease total system throughput, therefore the matchmaking should be parsimonious so as not to waste resources. Completeness means that as long as the system contains a node that satisfies a job's requirements, the matchmaker should find that node to run the job. Finally, the overall matchmaking process should not incur significant costs, to minimize overhead.

The ClassAd matchmaking framework is a flexible and general method of resource management in pools of resources which exhibit physical and ownership distribution. Aspects of the framework include a semi-structured data model to represent entities, folding the query language into the data model, allowing entities (resource providers and requestors) to publish queries as attributes. The paradigm also distinguishes between matching and claiming as two distinct operations in resource management: A match is an introduction between two compatible entities, whereas a claim is the establishment of a working relationship between the entities. The representation and protocols facilitate both static and dynamic heterogeneity of resources, which results in a robust and scalable framework that can evolve with changing resources.

The Content Addressable Network (CAN) is a distributed, decentralized P2P infrastructure that provides hash table functionality. The architectural design is a virtual multi-dimensional Cartesian coordinate space, a type of overlay network, on a multi-torus. Points within the space are identified with coordinates. The entire coordinate space is dynamically partitioned among all the nodes in the system such that every node possesses at least one distinct zone within the overall space.

A job in our system is the data and associated profile that describes a computation to be performed. The grid system may contain heterogeneous nodes with different resource types and capabilities, e.g. CPU speed, memory size, disk space, number of cores. Jobs submitted to the grid also can have multiple resource requirements, limiting the set of nodes on which they can be run. We assume that every job is independent, meaning that there is no communication between jobs. To build the P2P grid system, a variant of the CAN [17] distributed hash table (DHT) is employed, which represents a nodes resource capabilities (and a jobs resource requirements) as coordinates in the d-dimensional space. Each dimension of the CAN represents the amount of that resource, so that nodes can be sorted according to the values for each resource. A node occupies a hyper-rectangular zone that does not overlap with any other nodes zone, and the zone contains the nodes coordinates within the d-dimensional space. Nodes exchange load and other information with nodes whose zones abut its own (called neighbors). The following steps describe how jobs are submitted and executed in the grid system.

1) A client (user) inserts a job into the system through an arbitrary node called the injection node.
2) The injection node initiates CAN routing of the job to the owner node.
3) The owner node initiates the process to find a lightly loaded node (runnode) that meets all of the job's resource requirements (called matchmaking). (For more details on the owner node and matchmaking, refer to Kim et al. [8])
4) The run node inserts the job into an internal FIFO queue for job execution. Periodic heartbeat messages between the run node and the owner node ensure that both are still alive. Missing multiple consecutive heartbeats invokes a (distributed) failure recovery procedure.
5) After the job completes, the run node delivers the results to the client and informs the owner node that the job has completed.

The owner node monitors a job's execution status until the job finishes and the result is delivered to the client. To enable failure recovery, the owner node and the run node periodically exchange soft-state heartbeat messages to detect node failures (or a graceful exit from the system). More details about the basic system architecture can be found in Kim et al. [8]. The studies conducted in this paper can be used in any of the contexts discussed above or even any arbitrary network. Also, the waiting time is calculated as the non-executing time spent by the jobs after the job has migrated to the node where it would be scheduled for execution i.e. we do not account for the time spent by the job between the job submission and job migration in the network. This is in contrast to the waiting times usually computed in a distributed environment where it is the non-executing time spent by the job from the time it was submitted in the network until it completes execution. More on this is discussed in the 'Experiment and Results' section. The neighbors of the node are arbitrarily generated. We produce results for nodes with neighbors having similar resource constraints and nodes with larger number of neighbors in order to show the effectiveness of our algorithms in the CAN-like and other highly interconnected networks.

4 Term Representations

1) J_a = An arbitrary job in queue (Non executing job)
2) J'_a = Currently Running (or executing) Job
3) J_h = Job at head of queue
4) J'_{Pmin} = Minimum Priority Job running currently in a given set
5) J'_{Rmin} = Minimum Resource consuming Job running currently in a given set
6) $J_{covered}$ = Jobs covered so far for analysis
7) $J_{running}$ = Jobs currently running
8) J_{rem} = The remaining jobs (those yet to be examined for preemption)

9) P_{j_a} = Priority of Job J_a
10) $P_{j'_a}$ = Priority of currently running job
11) P_{j_h} = Priority of Job at head of queue J_H
12) $P_{max}(J_{covered})$ = Priority of the Highest priority job that is covered so far
13) $P_{min}(J_{covered})$ = Priority of the Lowest priority job that is covered so far
14) $P_{max}(J_{rem})$ = Priority of the Highest priority job from the remaining jobs (those yet to be examined for preemption)
15) $P_{min}(J_{rem})$ = Priority of the Lowest priority job from the remaining jobs (those yet to be examined for preemption)
16) R_{j_a} = Resource requirements for Job J_a. The algorithm treats all resource types (CPU's, GPU's, memory and disk space) as a set R.
17) R_f = Current free residual resources
18) $R_f(temp)$ = Residual resources that would be available when some current running jobs are preempted
19) R_{j_h} = Resource requirements of Job at head of queue
20) $R_{j'_i}$ = Resource requirements of Job currently running
21) $R_{j'_{Pmin}}$ = Resource requirements of J'_{Pmin}
22) $R_{j'_{Rmin}}$ = Resource requirements of J'_{Rmin}
23) $T_{rem}(J_a)$ = Remaining Time for Job J_a
24) W_{J_a} = Waiting time of Job J_a defined as the non-executing time spent by the job after it has migrated to the node where it would be scheduled for execution.

5 PREEMPTIVE SCHEDULING

5.1 Local Scheduling

This section deals with the scheduling criteria for a single node. As mentioned in section 2, we combine the ideas of 'shortest remaining time next' and the 'higher response ratio next' to come up with a preemptive scheduling algorithm for the grids. The 'shortest remaining time next' ensures that jobs that have the smaller remaining time are run, so they end sooner. However, this could lead to starvation for long running jobs and hence we increase the priority for jobs that wait longer in the queue. Thus, the jobs waiting in the node's queue have their priorities calculated as

$$P_{j_a} = ((\alpha * W_{J_a}) - (\beta * T_{rem}(J_a))$$

i.e. the priority for a job is directly proportional to its wait time W_{J_a} and negatively proportional to its estimated time for completion $T_{rem}(J_a)$. α is the weight factor associated with the wait time. β is the weight associated with the remaining time for completion. Typically, the β value is set to 1. The section on 'Experimental Results' provides more details on the values of α.

The job queue is sorted according to the order of their priorities calculated as above. Initially, the jobs in the head of the queue are scheduled until the available resources are insufficient for the next job to run. Next, those jobs that can

run in the available residual resources are scheduled to run (Backfilling). Since the backfilled jobs have priorities associated with them, they are also prone to preemption and therefore do not starve jobs waiting in the queue. The scheduler is invoked at the following 3 phases in the system:
1) After every periodic scheduling interval δ.
2) As a new job enters the queue.
3) A job completes its execution.

The periodic interval δ is much higher when compared to the scheduling intervals for schedulers in the OS. This is because in a heterogeneous environment we expect the time taken for context switches to be more expensive. And so, frequent context switches would result in low overall CPU utilization. More details regarding the values of δ are discussed in the Results section.

The scheduler is invoked when a new job enters the queue because the newly arrived job could be backfilled. And, when a job completes execution, it frees up some resources which allows new jobs to run. At every scheduling turn, the priority of the job in the head of queue is compared with that of the least priority job that is currently running. This is done because the queued job cannot run currently if its priority is lower than the lowest priority job that is currently running. This also addresses the backfilled jobs immediately since backfilled jobs have the lowest priority among the running jobs. Figure 1 demonstrates the scenario where the scheduler preempts a lower priority job with a higher priority job and backfills another job in its residual resources.

If the priority of the job at head of queue (P_{j_h}) is greater, the scheduler checks if the current running job J'_{Pmin} frees up enough resources for the new job to run. If yes, the job J'_{Pmin} is preempted and J_h is scheduled. Otherwise, the scheduler compares the priority of the second lowest priority job (J'_a) with J_h. This is carried out until the scheduler appropriately preempts jobs that free up just the right amount of resources for the job J_h to run. If the scheduler is unable to free up sufficient resources for the job to run, the job J_h is not scheduled in this interval and has to wait until the next scheduling turn. The scheduling (at every scheduling turn) is carried out for all jobs in the queue that have a higher priority than the lowest priority job that is currently running. The details are described in the 'Preemptive scheduling algorithm' below.

5.2 Context Switching and its impact

The cost of a context switch is well quantified in [11] and in (http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html). Although the paper talks about context switching time of up to $1.5ms$ for large working sets, the article in the blog gives a good worst case approximation for context switches (about 40 μs) for current Intel processors. Even if we assume a worst case value of $1ms$ for each context switch, that results in less than 0.02% error for scheduling interval $\delta = 5s$ in our calculations of wait times. In fact, we

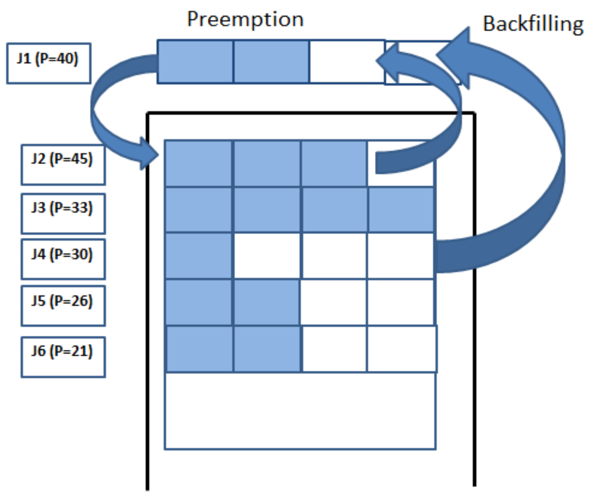


Fig. 1: Local Preemptive scheduling

only have context switches when there is preemption and so not every scheduling interval would have a context switch. The time taken for context switches can be further reduced if jobs are pinned to a particular core since this would avoid cache pollution (i.e. reduce the effect of thrashing). Due to the above reasons we believe it is safe to ignore the time taken for context switches in our experiments.

5.3 Internode Scheduling

Internode scheduling is an extended version of local scheduling; the target node for backfilling can be the neighboring nodes in the network. Local scheduling deals only with the changes to the job execution order within the queue on a node. Internode scheduling however, must decide the following:
1) Which node initiates job migration,
2) Which node should be the sender of a job,
3) and which job should be migrated.
Internode scheduling takes place periodically at every scheduling interval after the local scheduling process to see if the job at the top of the queue in the node can be run on any of its neighbors and also to see if the node can run the job of any of its neighbors in its currently free residual resources.

Algorithm 1

Preemptive Scheduling Algorithm

```
procedure
CalculatePriority(Job $j_a$)
 1: $P_{j_a} = ((\alpha * W_{J_a}) - (\beta * T_{rem}(J_a))$
end procedure
procedure
ScheduleJobs(JobQueue)
 1: Sort(JobQueue)
 2: while $R_f \neq 0$ do
 3:   $J_h$ = nextJobInQueue()
 4:   If $R_{j_h} < R_f$.
 5:   $R_f = R_f - R_{j_a}$
 6: end while
 7: $J_h$ = nextJobInQueue()
 8: while $(P_{j_h} > P_{min}(J_{running}) || P_{j_h} \neq 0)$ do
 9:   $J_{covered} = 0$
10:   $J_{rem} = 0$
11:   $R_f(temp) = R_f$
12:   $J_{covered} = J'_{Pmin}$
13:   if $R_{j_h} <= (R_{j'_{Pmin}} + R_f(temp))$ then
14:     Preempt($J'_{Pmin}$)
15:     Run($J_h$)
16:   else
17:     $R_f(temp) = R_f(temp) + R_{j'_{Pmin}}$
18:     FindJobstoPreempt()
19:   end if
20:   $J_h$ = nextJobInQueue()
21: end while
end procedure
procedure
FindJobstoPreempt( )
 1: while $(J_{rem} \neq 0)$ do
 2:   Select $J'_a$ such that $P_{j'_a} > P_{max}(J_{covered})$ and $P_{j'_a} = P_{min}(J_{rem})$
 3:   $J_{covered} += J'_a$
 4:   $J_{rem} = J_{running} - J_{covered}$
 5:   if $P_{j'_a} > P_{j'_h}$ then
 6:     break {cannot preempt jobs}
 7:   else
 8:     if $R_{j_h} <= R_{j'_a}$ then
 9:       Preempt($J'_a$)
10:       Run($J_h$)
11:       break
12:     else
13:       FindOptimal($J_{covered}, J_h$)
14:     end if
15:   end if
16: end while
end procedure
```

```
procedure
FindOptimal($J_{covered}, J_h$)
1: if $R_{j_h} <= (R_{j_{a'}} + R_f(temp))$ then
2:   for each $J'[i]$ in $J'_{covered}$ with $P_{j'_i} < P_{j'_a}$ and $R_{j'[i]} = R_{J'_{Rmin}}$
3:   if $R_{j'_a} + R_{j'[i]} >= R_{j_h}$ then
4:     Preempt $J'[i]$ ,$J'_a$
5:     Run $J_h$
6:     break
7:   else
8:     searchCombinationsforOptimalPreemption($J'[i], J'_{covered}$)
9:   end if
10:  end for
11: else
12:   $R_f(temp) = R_f(temp) + R_{j'_a}$
13: end if
end procedure
```

In the PUSH scheduling model the job sender initiates the migration process. First, the sender node tries to match priority of the job at the head of the queue with the neighboring node's queue. If the priority of the job at head of the queue in its neighbor node is less than the job at the sender node, a PUSH message for the job is sent to its neighbor containing the job's priority (of sender node) and the resource requirements. If the job can be backfilled at the neighbor node, the PUSH message is accepted. Otherwise, a PUSH-reject message is sent back to the sender node. If a job can be run on multiple neighbors, the sender sends it to the node that has minimum objective function value as follows. Figure 2 shows the case where a job at the head of queue on one node is pushed to run on the neighboring node.

$$f_{Inter-PUSH} = BM * FM * (1/CPU_{speed})$$

where BM and FM are defined as follows:

$$BM = \frac{max_k(S^k + R_j^k)}{\frac{(\sum_{k=1}^{K}(S^k+R_j^k)}{K}} = \frac{MaximumUtilization}{AverageUtilization}$$

$$FM = 1 - \frac{(\sum_{k=1}^{K}(S^k+R_j^k)}{K} = 1 - AverageUtilization$$

where K is the number of resources (or requirements), S^k is normalized utilization for resource $k(1 < k < K, 0 < S^k < 1)$, and R_j^k is job j's normalized requirement for resource $k(0 < R_j^k < 1)$. BM measures unevenness across utilization of multiple resources, and FM measures how much resources are underutilized on average. Therefore, lower BM and FM imply better balanced resource utilization and better average utilization, respectively.

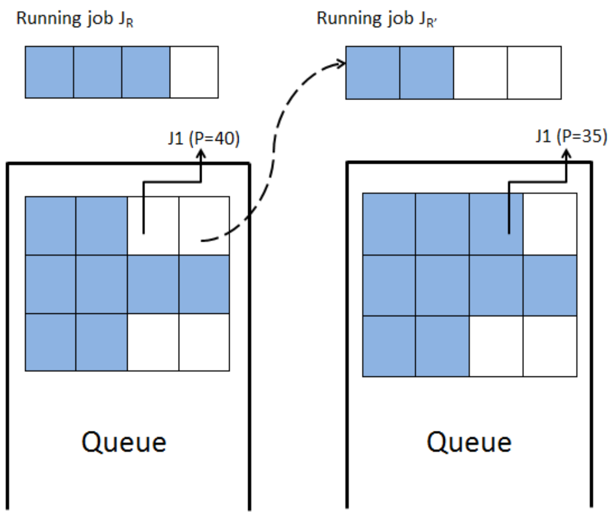


Fig. 2: Internode scheduling

To prefer the fastest node among neighbors, the objective function also includes an inverse term for CPU speed. Before sending a job profile, there is a simple confirming handshake process between a sender and a potential receiver to avoid inappropriate job migration because the potential receiver information may not be up-to-date at the sender.

In the PULL model, a receiver node tries to obtain a job from its CAN neighbors so as not to waste its available resources. However, the node does not have all information on the queued jobs resource requirements in its neighbors to minimize neighbor update message sizes, so the node invokes a PULL-Request message to the node having the closest priority job at the head of queue that is higher than the priority of job at the head of the queue in the current node. If there are multiple such nodes, the request is sent to the node with maximum queue size among its neighbors. If there are multiple candidate jobs in the waiting queue, then the job that has minimum objective function value (BM * FM, as above), is selected. If there is no candidate job, then the requesting node gets a PULL-Reject message and continues to look for another potential sender having the appropriate priority along with maximum queue length not contacted recently.

6 Experiment and Results

6.1 Experimental Setup

A synthetic workload was generated to model the grid resource configuration containing heterogeneous nodes capable of executing a heterogeneous set of jobs.

The simulation scenario consists of 1000 multi-core nodes (having 1, 2, 4 or 8 cores), and 5000 jobs submitted to run on those nodes. Each node has multiple resource capabilities such as CPU speed, memory size, disk space and the number of cores. The jobs are also modeled similarly having the heterogeneous resource configuration as their requirements. A high percentage of the nodes (and jobs) have relatively low resource capabilities (requirements), and a low percentage of nodes (and jobs) have high resource capabilities (requirements).

The interval between job submissions follow a Poisson distribution, with varying average job inter arrival times in the experiments. Each job has an estimated running time associated with it. The estimated times are uniformly distributed between 0.5T and 1.5T, with T= 3600 seconds, running on a canonical node with a normalized CPU speed of 1. The simulated job running time is then scaled up or down by the CPU speed relative to the canonical node.

We compare our schemes to the FCFS scheduler with backfilling which schedules jobs in the order they arrive and also performs backfilling of jobs on residual resources. To measure the performance of the long running grid system, we run the simulations in a steady state environment. By steady state, its implied that the job arrival and departure rates are similar, so that the system achieves a dynamic equilibrium state during the simulation period, with the system neither highly overloaded nor underutilized. Hence, the average total system load is determined by the inter-job arrival rate. However, very lightly loaded systems were not tested, because they are not very interesting for measuring dynamic scheduling performance.

The total waiting time for a job is usually calculated as the non-executing time spent by the job from the time it was submitted in the network till it completes execution. However, in this paper we do not account for the time spent by the job between the job submission and job migration process in the network. Instead we consider the job arrival time as the time at which the job arrives at the node where it can be executed. Thus, the wait times are redefined as the non-executing time spent by the jobs after the job has migrated to the node where it would be scheduled for execution.

The neighbors of the node are arbitrarily generated. We produce results for nodes with neighbors having similar resource constraints and nodes with varying neighbors in order to show the effectiveness of our algorithms in the CAN and other interconnected networks. Specifically, we produce results for a network where each node is connected to exactly two other nodes (abbreviated as 2-NN) and a CAN-like network with 3-4 neighbors. We say CAN-like because the network constructed does not strictly adhere to CAN specifications though each node is connected to 4 other nodes that have similar resource capabilities. For simplicity we refer to the CAN-like network as CAN' in the following sections.

6.2 Experimental Results

Figure 3 lists and compares the median wait times across all jobs for each job inter-arrival time. The median wait times are plotted for the four scenarios FCFS with Backfilling, Local preemptive scheduling with Backfilling and Internode scheduling (in both CAN' and 2-NN). We experimented with different values for α. However, setting $\alpha = 0$ yielded the lowest median wait times across all jobs and so, we use this value to plot our graphs. This is essentially a Shortest Job First preemptive strategy i.e. at any time, the job with the smallest remaining executing time is chosen to run irrespective of its waiting time in the queue.

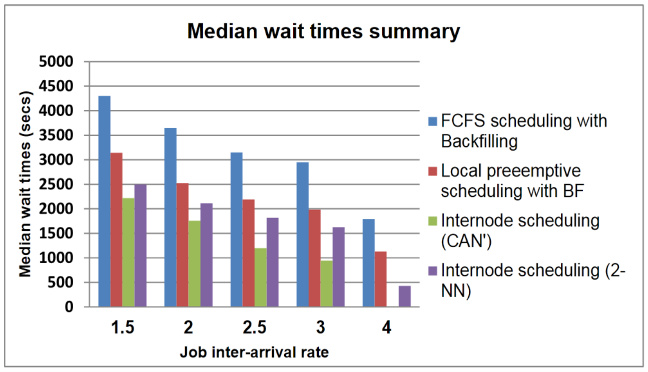
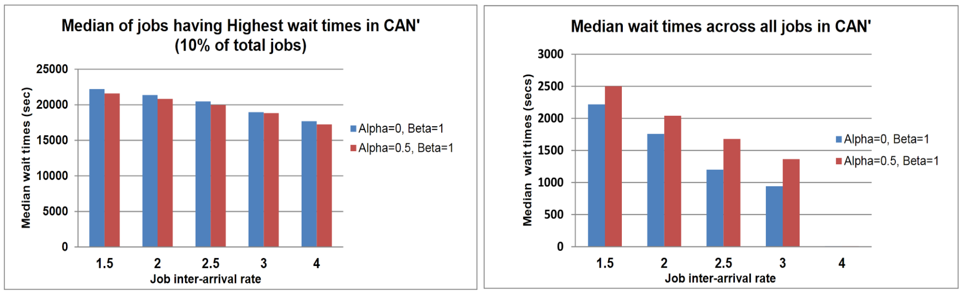


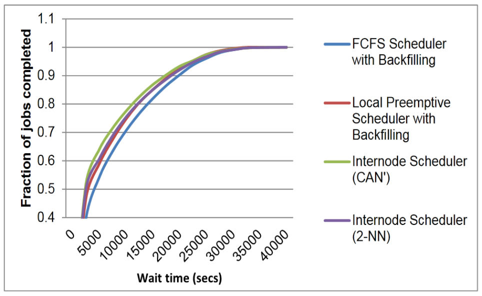
Fig. 3: Median wait times for different Job inter-arrival times



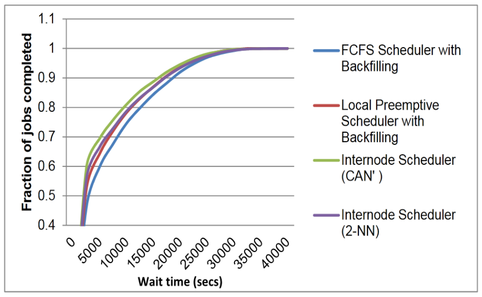
(a) Median waittimes in long waiting jobs (b) Overall median waittimes

Fig. 4: Median wait time comparisons in CAN'

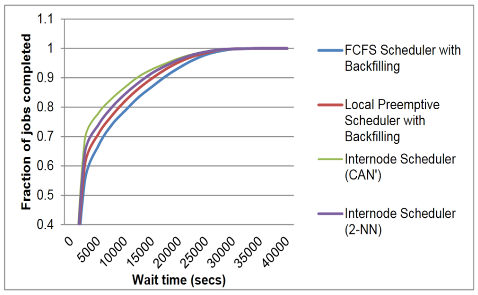
When the jobs have low inter-arrival times, jobs arrive quickly onto the node and spend more time waiting in the queue. In contrast, when jobs have higher inter-arrival times, they arrive considerably later than its previous job and end up with comparatively lower wait times. It is clear from Figure 3 that local preemptive scheduling algorithm results in significantly lower wait times when compared to the FCFS strategy for all cases of job inter-arrival times. Significant differences can also be observed between waiting times of local preemptive and internode scheduling proving the effectiveness of the internode scheduling algorithm. The differences in wait times for CAN' and 2-NN internode scheduling algorithms is low for low job inter-arrival times (1.5 and 2.0) and increases with increase in job inter-arrival times. This shows that the internode scheduling is more effective for more neighbors especially when the job inter-arrival time is high. This is because for low job inter-arrival times, job migrations to neighboring nodes are rare since those nodes are already executing many jobs.



(a) Inter-arrival rate 1.5



(b) Inter-arrival rate 2.5



(c) Inter-arrival rate 4.0

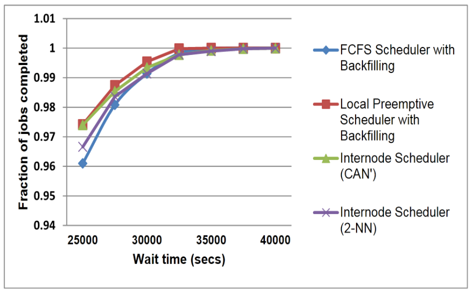
Fig. 5: Fraction of jobs completed in the four schedulers

We also conducted experiments for values $\alpha = 0.5$ and $\beta = 1$ so that jobs that have been waiting in the queue for a while get a chance to run. In this scenario, the jobs that have waited in the queue for a long time compete against the shorter running jobs. The intuition behind this experiment was to prevent the long waiting jobs from being starved of CPU and to reduce their total waiting time. Figure 4a shows the gain achieved in the wait times for the top 10% of long waiting jobs in CAN'. We also observed similar results for the 2-NN and local preemptive scheduling scenarios. Fig 4b shows the median wait time comparisons of the two approaches. The choice of picking the appropriate value of alpha depends on what type of service we intend to provide the end-users (i.e. bounded wait times for all jobs vs. highest throughput for most jobs). We believe the values of the wait times to be dependent on the type of load (jobs) being submitted to the nodes and the network environment.

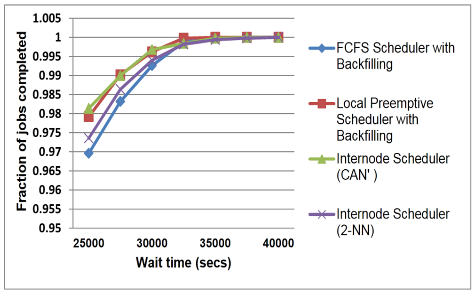
Figures 5a to 5c illustrates the distribution of the wait times for jobs in all the environments, i.e. preemptive scheduling (both local and internode scheduling) and non-preemptive FCFS scheduling. The first 2000 jobs having the lowest wait times have been omitted in plotting the graphs. We did this because so many jobs wait for very little time and so cutting off the part where all the lines completely overlap doesn't lose any information. The plots show that the waiting times of jobs decreases with increasing job inter-arrival times in the FCFS environment. The curves in Figure 5a show improvement in the percentage of jobs completed with low wait times for local and internode scheduling as compared to the FCFS scheduling. We can also observe that the curves for local preemptive scheduling and internode scheduling (for 2-NN) are almost overlapping. However, the distinction between these curves becomes more apparent with higher job inter-arrival times. We can see a marked improvement (in Figure 5c) on the percentage of jobs completed with low wait times for our preemptive scheduling strategies over the non-preemptive FCFS approach when the job inter-arrival rate is 4.0. The internode scheduling in CAN' performs significantly better than the non-preemptive FCFS strategy.

We also repeated the same experiment for a smaller scheduling interval of 2.5 sec to observe any significant variances in the wait times. However, the improvements in the median and average wait times were almost negligible except for Internode-scheduling for inter-arrival rate of 4 seconds in CAN'. The CAN' (for inter arrival time=4s) responded very well (almost 50% decrease in median wait time) with the change in the scheduling interval. We think this is because the CAN' has more neighboring nodes with similar resource requirements that are capable of running the job and thus succeeds with a higher probability of scheduling the job when compared to the 2-NN topology. Also, due to high job inter-arrival time the scheduler is able to find more such nodes because the neighboring nodes have more likelihood of having empty cores. We believe that factors such as the order in which the jobs are submitted, their execution times and the resource requirements for these jobs, all play a critical role in determining the optimal scheduling interval. More on this is discussed in the Future Work section.

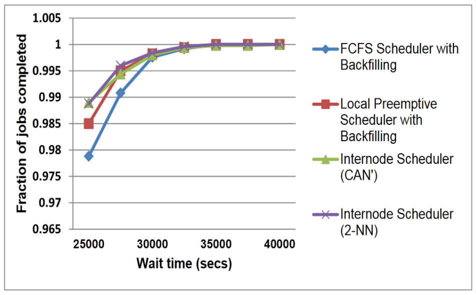
Another important scheduling criterion is reducing the maximum wait time, so that no (or fewer) jobs wait a very long time to run. Figures 6a to 6c focuses on the tails of the job distributions of Figures 5a to 5c (the last 100-200 jobs having the highest wait times). Figure 6a shows that the local preemptive scheduler does better than internode scheduler when the job inter-arrival rate is 1.5. We believe this is because since a large number of jobs arrive in a short span of time, the jobs migrated to neighboring nodes would result in longer job queues for some nodes; thus increasing the wait times for jobs further down the queue. In other words, there is a load imbalance. We can see a similar trend in Figure 6b though there are fewer such jobs. As the job inter-arrival time increases, this effect is reduced. In Figure 6c we observe an interesting trend where Internode-2NN does better than both local preemptive and Internode CAN' schedulers.



(a) Inter-arrival rate 1.5



(b) Inter-arrival rate 2.5



(c) Inter-arrival rate 4.0

Fig. 6: Fraction of jobs completed in the four schedulers (towards the end)

The total number of preemptions for shortest-job first strategy in the Local preemptive scheduling (for scheduling interval of 5 secs) scenario varied from 556 (for Inter-arrival times=1.5 secs) to 471 (for IAT = 4 secs) while that for Internode scheduling varied between 570 to 480. This was approximately equal to 1/8th the total number of jobs submitted in the system. As mentioned before, we believe that factors such as the order in which the jobs are submitted, their execution times, resource requirements and load balancing of these jobs all play a critical role in determining these numbers. For scheduling interval (δ) of 2.5

secs, we didnt notice any significant differences in these values. The number of preemptions started to increase considerably only when the value of alpha was set to 1 or higher. But this resulted in high median and average wait times across the network.

7 Conclusion

A preemptive scheduling algorithm (with backfilling) for multi-core grid resources was designed and implemented. As part of local scheduling, jobs that are estimated to complete sooner were given higher priority compared to long running jobs while at the same time ensuring that the long running jobs get their fair share of the CPU. The results show that our algorithm yields lower average and median wait times when compared to the FCFS approach. In particular, the shortest-job first algorithm yields the lowest median wait-times for the system compared to cases where long running jobs compete for the CPU. The Internode scheduling ensures that those jobs that cannot be immediately scheduled are PUSHED to a neighboring node if it can run in their residual resources. It also allows a node to PULL jobs from neighboring nodes to utilize its local residual resources. An appropriate value for α, the weight for the waiting time for a job, ensures to lower the wait times for long waiting jobs. In addition, the median wait times for CAN-like systems can be further lowered by choosing the appropriate value for the scheduling interval δ.

8 Future Work

The local scheduling and internode scheduling algorithms find and execute a job using residual free resources in a node. This means that only jobs that can start running immediately will be moved. However, if the load across nodes is skewed, the job queue lengths vary greatly, and hence a more pro-active queue balancing scheme would improve load distribution and overall throughput across heterogeneous nodes. To address this, we illustrate the same technique used in [9] here. Firstly, the maximally loaded resource among the K available resources is set as the Load of a node, and the algorithm minimizes the total sum of the Loads among neighbors, and also balances Load across the nodes[22]. The term W_i^k is defined, normalized load for Resource k of Node i by:

$W_i^k = \sum_{J_j \in Queue_i} (R_j^k), 1 \leq k \leq K$

where J_j is Job j, R_j^k is the kth normalized resource requirement for J_j, and $Queue_i$ is the job queue for node i. The normalized load of Node i, L_i is given by

$L_i = Max(W_k^i), 1 \leq k \leq K$

The PUSH and PULL job migration models can be used for queue balancing,

as they were for internode scheduling. For PUSH, a node i computes normalized load (L_i) for itself and for its neighbors. If L_i is the locally maximum value among all its neighbors, then node i checks its queue to find candidate jobs for migration that reduce L_i if the (candidate) job is moved. Among these jobs, those jobs that satisfy the priority constraints in the neighboring node are considered. When there are multiple candidate jobs, the algorithm selects the job and the receiver node that minimize an objective function if the job is moved to the neighbor.
The PULL model is similar to the PUSH model, except that the node with a locally non-zero minimum normalized load among equal or less capable neighbors will initiate the PULL process from the most loaded node among its neighbors. The Queue Balancing technique may further improve the performance of the desktop grid system.
Further research can be done by experimenting with different sets of workloads for different types of networks. We could then observe what values of δ and α give optimal values for the median wait times across the nodes.

9 Acknowledgements

We appreciate the comments received from anonymous reviewers of the JSSPP 2013 workshop. They pointed out some key issues that has led us to do further research on this topic. We thank Manjunath Gopinath, Sarat Babu Eruvuru, Bhavani Bhaskar and Abhishek Prasad for their participation in discussions and their feedback on this idea.

References

1. I. Al-Azzoni and D. G. Down. Dynamic scheduling for heterogeneous desktop grids. *Journal of Parallel and Distributed Computing*, 70(12):1231–1240, December 2010.
2. M. J. Bach. *The Design of the UNIX Operating System, Chapter 8 - Process Scheduling and Time.* Prentice Hall.
3. F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogenous networks. In *Proceedings of the 1996 ACM/IEEE Conference on SuperComputing.* ACM/IEEE, 1996.
4. D.Zhou and V.Lo. Wave scheduler: Scheduling for faster turnaround time in peer-based desktop grid systems. In *Proceedings of the 11th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2005.
5. D.Zhou and V.Lo. Wavegrid: a scalable fast-turnaround heterogeneous peer-based desktop grid system. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS2006).* IEEE Computer Society Press, April 2006.
6. M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 21(2):207–233, 2003.
7. V. J. Jimenez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09*

Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers, January 2009.

8. J.S.Kim, P.Keleher, M.Marsh, B.Bhattacharjee, and A.Sussman. Using content-addressable networks for load balancing in desktop grids. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC-16)*, June 2007.
9. J. Lee, P. Keleher, and A. Sussman. Decentralized dynamic scheduling across heterogeneous multi-core desktop grids. In *Proceedings of the 19th International Heterogeneity in Computing Workshop (HCW2010).* IEEE Computer Society Press, April 2010.
10. J. Lee, P. Keleher, and A. Sussman. Supporting computing element heterogeneity in p2p grids. In *Proceedings of the IEEE Cluster 2011 Conference.* IEEE Computer Society Press, September 2011.
11. C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science ExpCS '07.* ACM New York, 2007.
12. M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, 1988.
13. S. Niu, J. Zhai, X. Ma, M. Liu, Y. Zhai, W. Chen, and W. Zheng. Employing checkpoint to improve job scheduling in large-scale systems. In *16th International Workshop on Job Scheduling Strategies for Parallel Processing, JSSPP 2012, Shanghai, China*, pages 36–55, May 2012.
14. R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughput computing. In *Proceedings of the 7th International Symposium on High Performance Distributed Computing.*, pages 140–146, July 1998.
15. S.Moore. Multicore is bad news for super computers. *IEEE Spectrum.*, 45(11):15–15, November 2008.
16. Q. Snell, M. J.Clement, and D. B. Jackson. Preemption based backfill. In *JSSPP '02 Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 24– 37, 2002.
17. S.Ratnasamy, P.Francis, M.Handley, R.Karp, and S.Shenker. A scalable content addressable network. In *Proceedings of the ACMSIGCOMM Conference*, August 2001.
18. W. Stallings. *Operating systems: internals and design principles. 4th ed.* Prentice Hall, ISBN 0-13-031999-6, 2001.
19. A. S. Tanenbaum. *Modern Operating Systems (3rd ed.).* Pearson Educationl, ISBN 0-13-600663-9, 2008.
20. K. Thompson. *UNIX Implementation.* Bell Laboratories.
21. W.Leinberger, G.Karypis, and V.Kumar. Job scheduling in the presence of multiple resource requirements. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing(CDROM).*, page 47, NewYork, NY, USA, 1999. ACM.
22. W.Leinberger, G.Karypis, V.Kumar, and R.Biswas. Load balancing across near-homogeneous multi-resource servers. In *Proceedings of the 9th Heterogeneous Computing Workshop, appears with the Proceedings of IPDPS 2000*, pages 60–71, 2000.
23. F. Xhafa and A. Abraham. Computational models and heuristic methods for grid scheduling problems. *Future Generation Computer Systems*, 26(4):608–621, April 2010.