A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment*

Yanyong Zhang Anand Sivasubramaniam Jose Moreira Hubertus Franke

Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802
{yyzhang,anand}@cse.psu.edu

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
{jmoreira,frankeh}@us.ibm.com

ABSTRACT

Scheduling of processes onto processors of a parallel machine has always been an important and challenging area of research. The issue becomes even more crucial and difficult as we gradually progress to the use of off-the-shelf workstations, operating systems, and high bandwidth networks to build cost-effective clusters for demanding applications. Clusters are gaining acceptance not just in scientific applications that need supercomputing power, but also in domains such as databases, web service and multimedia, which place diverse Quality-of-Service (QoS) demands on the underlying system. Further, these applications have diverse characteristics in terms of their computation, communication and I/O requirements, making conventional parallel scheduling solutions, such as space sharing or coscheduling, an unattractive option. At the same time, leaving it to the native operating system of each node to make decisions independently can lead to ineffective use of system resources whenever there is communication. Instead, an emerging class of dynamic coscheduling mechanisms, that attempt to take remedial actions to guide the system towards coscheduled execution without requiring explicit synchronization, offer a lot of promise for cluster scheduling. Using a detailed simulator, this paper evaluates the pros and cons of different dynamic coscheduling alternatives, while comparing their advantages over traditional coscheduling (and not performing any coordinated scheduling at all). The impact of dynamic job arrivals, job characteristics and different system parameters on these alternatives are evaluated in terms of several performance criteria.

Keywords: Parallel Scheduling, Coscheduling, Dynamic Coscheduling, Clusters, Simulation.

*This research has been supported in part by NSF Career Award MIP-9701475, grant CCR-9900701, and equipment grants CDA-9617315 and EIA-9818327.

1. INTRODUCTION

Scheduling of processes onto processors of a parallel machine has always been an important and challenging area of research. Its importance stems from the impact of the scheduling discipline on the throughput and response times of the system. The research is challenging because of the numerous factors involved in implementing a scheduler. Some of these influencing factors are the parallel workload, presence of any sequential and/or interactive jobs, native operating system, node hardware, network interface, network, and communication software. The recent shift towards the adoption of off-the-shelf clusters/networks of workstations (called COWs/NOWs) for cost-effective parallel computing, makes the design of an efficient scheduler even more crucial and challenging. Clusters are gaining acceptance not just in scientific applications that need supercomputing power, but also in domains such as databases, web service and multimedia, which place diverse Quality-of-Service (QoS) demands on the underlying system (not just higher throughput and/or lower response times). Further, these applications have diverse characteristics in terms of the computation, communication and I/O operations which raises complications in multiprogramming the system. Traditional solutions that have been used in conventional parallel systems are not adequately tuned to handle the diverse workloads and performance criteria required by cluster environments. This paper investigates the design space of scheduling strategies for clusters by extensively evaluating nine different alternatives to understand their pros and cons, and compares them with a conventional solution (coscheduling).

Scheduling is usually done in two steps. The first step, spatial scheduling, consists of assigning processes to nodes. (A node can have one or more processors, and runs a single operating system image.) The second step, temporal scheduling, consists of time multiplexing the various processes assigned to a node for execution by the processors of that node. There is a considerable body of literature regarding spatial scheduling and we do not delve on this problem in this paper, nor do we examine the issue of migrating processes during execution for better load balance. Without loss of generality, for this paper we assume one processor (CPU) per node, and an incoming job specifies how many CPUs it needs, and executes one task (process) on each of the allocated CPUs.

The second scheduling step, temporal scheduling, is perhaps more important for a cluster. Just as common off-the-shelf (COTS) hardware has driven the popularity of clusters, it is rather tempting to leave it to the native (COTS) operating system scheduler to take care of managing the processes assigned to its node. However, the lack of global knowledge at each node can result in lower CPU utilization, and higher communication or context switching overheads. As a result, there have traditionally been two approaches to address this problem. The first is space sharing [8], which is a straightforward extension of batching to parallel systems. Multiple jobs can execute concurrently at different nodes, but each node is exclusively allocated to one process of a job, which then runs to completion. Space sharing is simple to implement, and reduces context switching costs. However, space sharing in isolation can result in poor utilization (nodes can be free even though there are jobs waiting). Also, the lack of multiprogramming can hurt when a process performs a lot of I/O. The second approach is a hybrid scheme, called (exact) coscheduling or gang scheduling [12, 5, 4, 6], that combines time sharing with space sharing. The processes of a job are scheduled on their respective nodes at the same time for a given time quantum, the expiration of which results in a synchronization between the nodes (using logical or physical clocks) to decide on the next job to run. This scheme usually requires long time quanta to offset high context switching and synchronization costs. Longer time quanta make the system less responsive for interactive and I/O intensive jobs (database services, graphics and visualization applications etc.). In addition, strict coscheduling also keeps the CPU idle while a process is performing I/O within its allotted quantum [6].

Recently, there has been interest in developing strategies that approximate coscheduling behavior without requiring explicit synchronization between the nodes (that still combine space and time sharing). We refer to this broad class of strategies that approximate coscheduled execution as *dynamic coscheduling mechanisms*. The enabling technologies that have made this approach possible is the ability of the network interface card (NIC) and messaging layers to provide protected multi-user access to the network in conjunction with user-level messaging. It is not necessary to perform network context switching when the processes are switched out on their respective CPUs as was necessary [17, 7] until recently. Dynamic coscheduling strategies try to hypothesize what is scheduled at remote nodes using local events (messaging actions/events in particular), to guide the native operating system scheduler towards coscheduled execution whenever needed. These strategies offer the promise of coscheduling, without the related overheads and scalability/reliability problems.

Prior to our work [10, 11], there were only two suggestions [1, 15, 2] on how local messaging actions can be used to implement dynamic coscheduling. Both these mechanisms incur interrupts which can hurt performance under some situations. We have proposed two alternates called Periodic Boost and Spin Yield, and have experimentally shown Periodic Boost to outperform the rest using a cluster of eight Sun Ultra Enterprise servers running MPI applications [10, 11]. While our earlier study is a preliminary foray into this area, a comprehensive exercise exploring the pros and cons of these different alternatives is needed to answer several open and crucial questions:

- How do the different dynamic coscheduling alternatives compare when one considers dynamic job arrivals with different job sizes (number of CPUs) and execution times? Our earlier exercise considered only a few (constant) pre-determined number of jobs, each demanding a constant number of CPUs and taking the same execution time. How does the arrival rate (load) of the jobs affect the average response times and throughput of the system?

- What is the impact of job characteristics on the performance of the system for the different scheduling alternatives? Specifically, how do the schemes compare as one varies the relative fraction of the computation (requiring CPU and memory resources only), communication and I/O performed by a job? What is the impact of a multiprogrammed workload consisting of different job mixes? As mentioned earlier, clusters are intended to take on the demands of diverse applications, each with its own computation, communication and I/O characteristics (for instance, a database application may be I/O intensive while a scientific application may be CPU or communication intensive), and should still meet the QoS requirements of each application. In addition, how does the work imbalance and skewness between the tasks (executing on different CPUs) affect the performance of each alternative?

- How do the system parameters such as the multiprogramming level at each node and the operating system costs for context switching and interrupt processing affect the relative performance of the schemes?

We attempt to answer these questions using an extensive simulation framework and an abstraction of a real workload [6] that has been drawn from an actual supercomputing environment (Lawrence Livermore National Labs). Eight different dynamic coscheduling strategies are evaluated using this infrastructure, and compared with exact co-scheduling (as well as with not performing any coordinated scheduling at all) to draw revealing insights. To our knowledge, there has not been any prior work exploring this extensive design space using a spectrum of performance metrics (throughput, response time, wait time, and utilization) and dynamic workloads.

The rest of this paper is organized as follows. The next section explains the different scheduling alternatives and how they are modeled in our simulator. Section 3 gives details on the simulator itself, together with the simulation parameters and performance metrics under consideration. The performance results are presented in Section 4. Finally, Section 5 summarizes the observations and outlines directions for future research.

2. SCHEDULING STRATEGIES

In the following discussion, we give a quick overview of user-level messaging. We then present the (exact) coscheduling model followed by the native operating system scheduler at

each node that is modeled as the core around which the dynamic coscheduling mechanisms are structured. Finally, the details of the different coscheduling heuristics are presented. All the models have been designed and developed based on our implementation [10, 11] of these mechanisms on an actual Sun Solaris cluster connected by Myrinet.

2.1 User-level Networking (ULN)

Traditional communication mechanisms have necessitated going via the operating system kernel to ensure protection. Recent network interface cards (NIC) such as Myrinet, provide sufficient capabilities/intelligence, whereby they are able to monitor regions of memory for messages to become available, and directly stream them out onto the network without being explicitly told to do so by the operating system. Similarly, an incoming message is examined by the NIC, and directly transferred to the corresponding application receive buffers in memory (even if that process is not currently scheduled on the host CPU). From an application's point of view, sending translates to appending a message to a queue in a region of its virtual address space (called an *endpoint*), and receiving translates to (waiting and) dequeuing a message from its endpoint. To avoid interrupt processing costs, the waiting is usually implemented as polling (busy-wait). Experimental implementations of variations of this mechanism on different hardware platforms have demonstrated end-to-end (application-to-application) latencies of 10-20 microseconds for short messages [18, 19, 13], while most traditional kernel-based mechanisms are an order of magnitude more expensive.

User-level messaging, though preferable for lowering the communication overhead, actually complicates the issue from the scheduling viewpoint. A kernel-based blocking receive call, would be treated as an I/O operation, with the operating system putting the process to sleep. This may avoid idle cycles (which could be given to some other process at that node) spent polling for message arrival in a user-based mechanism. Efficient scheduling support in the context of user-level messaging thus presents interesting challenges.

2.2 Coscheduling or Gang Scheduling (GS)

Exact coscheduling or Gang Scheduling (we will henceforth refer to this as just coscheduling) ensures that the processes/tasks of a job are scheduled on their respective nodes at the same time. This usually requires some means of explicit or implicit synchronization to make a coordinated scheduling decision at the end of each time quantum.

The simulation model is based on the implementation of the GangLL scheduler [9, 6] on the Blue Pacific machine at Lawrence Livermore National Labs. The model uses an Ousterhout [12] matrix with the columns representing the CPUs and rows representing the time quanta (as many rows as the multiprogramming level). In an actual system, the multiprogramming level (MPL) will be set based on the available resources (such as memory, swap space etc.) that can handle a certain number of jobs concurrently without significantly degrading performance. A job is allocated the required number of cells in a single row if available. Else, it is made to wait in an *arrival queue* (served in FCFS order) until there are enough free cells in a row. During each time quantum, a CPU executes the assigned job for that row in the matrix, and does not move to the next row until the next quantum (regardless of whether the process is waiting for a message, or performing I/O, or even finishes before the quantum ends). At the end of the quantum, a context switch cost is incurred. This not only includes the traditional costs, but also the cost for synchronizing between the nodes before it schedules the job for the next quantum (GangLL [6] actually uses physical clocks with large time quanta instead of explicit synchronization). Message receives are implemented as busy-waits (spinning), though some of this time could get hidden if the process is context switched out (quantum expires).

2.3 Local Scheduling

We refer to the system which does not make any coordinated scheduling decisions across the nodes as *local* scheduling. The native operating system is left to schedule the processes at each node. As in coscheduling, each node can again handle a maximum of MPL processes at any time, with the difference that an arriving job does not have to wait until free slots are found in a single row. Rather, a job can be scheduled to the corresponding CPUs that are not already operating at their full MPL capacity (can be a different row position for each column if one is to look at this problem as filling the Ousterhout matrix) . If the job cannot find that many CPUs, it waits in an *arrival queue* (served in FCFS order) until it does. A brief description of the native scheduler (multi-level feedback queue) at each node, which closely resembles the Solaris scheduler, follows.

There are 60 priority levels (0 to 59 with a higher number denoting a higher priority) with a queue of runnable processes at each level. The process at the head of the highest priority queue is executed first. Higher priority levels get smaller time slices than lower priority levels, that range from 20 ms for level 59 to 200 ms for level 0. At the end of the quantum, the currently executing process is degraded to the end of the queue of the next lower priority level. Process priority is boosted (to the head of the level 59 queue) when they return to the runnable state from the blocked state (completion of I/O, signal on a semaphore etc.) This design strives to strike a balance between compute and I/O bound jobs, with I/O bound jobs typically executing at higher priority levels to initiate the I/O operation as early as possible. The scheduler, which runs every millisecond, ensures that lower priority processes are preempted if a higher priority process becomes runnable (the pre-emption may thus not take place immediately after priority changes). For fairness, the priorities of all processes are raised to level 59 every second.

The ULN messaging actions explained above are used as is in *local*; send is simply an append to a queue in memory and receive is busy waiting (spinning) in user-space for message arrival (consuming CPU cycles) This scheme has been considered as a baseline to show the need for a better scheduling strategy.

2.4 Dynamic Coscheduling Strategies

As mentioned earlier, these strategies rely on messaging actions to guide the system towards coscheduled execution, and there is no coordinated effort explicitly taken to achieve this goal. Logically, there are two components in the interaction between a scheduler and the communication mech-

anism. The first is related to how the process waits for a message. This can involve: (a) just spinning (busy wait); (b) blocking after spinning for a while; or (c) yielding to some other process after spinning for a while. The second component is related to what happens when a message arrives and is transferred to application-level buffers. Here again, there are three possibilities: (a) do no explicit rescheduling; (b) interrupt the host and take remedial steps to explicitly schedule the receiver process; and (c) periodically examine message queues and take steps as in (b). These two components can be combined to give a 3×3 design space of dynamic coscheduling strategies as shown in Table 1.

What do you do on message arrival?	How do you wait for a message?		
	Busy Wait	Spin Block	Spin Yield
No Explicit Reschedule	Local	SB	SY
Interrupt & Reschedule	DCS	DCS-SB	DCS-SY
Periodically Reschedule	PB	PB-SB	PB-SY

Table 1: Design space of Dynamic Coscheduling strategies

In the following discussion, we limit our explanations to familiarize the reader with these strategies and to explain how they are simulated, rather than give a detailed discussion of their implementation on an actual operating system. For a detailed description of the implementation of these different strategies on a Sun Solaris cluster connected by Myrinet, the reader is referred to [10]. The simulation models and parameters are based on our earlier experimental exercises. All these strategies use the same scheme described above in *local* to assign the processes (tasks) of an arriving job to the different CPUs.

2.4.1 Spin Block (SB)

Versions of this mechanism have been considered by others in the context of implicit coscheduling [3, 1] and demand-based coscheduling [15]. In this scheme, a process spins on a message receive for a fixed amount of time before blocking itself. The fixed time for which it spins, henceforth referred to as *spin time*, is carefully chosen to optimize performance. The rationale here is that if the message arrives in a reasonable amount of time (spin time), the sender process is also currently scheduled and the receiver should hold on to the CPU to increase the likelihood of executing in the near future when the sender process is also executing. Otherwise, it should block so that CPU cycles are not wasted.

The simulation model sets the spin time for a message slightly higher than the expected end-to-end latency (in the absence of any contention for network or node resources) of the message it is waiting for. If the corresponding message arrives within this period, the mechanism works the same way as the earlier scheduling schemes (busy-wait). Else, the process makes a system call to block using a semaphore operation. On subsequent message arrival, the NIC firmware (having been told that the process has blocked) raises an interrupt, which is serviced by the kernel to unblock the process. As mentioned earlier, the process gets a priority boost (to the head of the queue of the highest priority level) on wakeup. Costs for blocking/unblocking a process, context switches resulting from these operations, and interrupt processing are modeled in the simulation.

2.4.2 Spin Yield (SY)

In SB, the process blocks after spinning. This has two consequences. First, an interrupt is required to wake the process on message arrival (which is an overhead). Second, the block action only relinquishes the CPU and there is no hint given to the underlying scheduler as to what should be scheduled next. In our earlier work [10], we have proposed Spin Yield (SY) as an alternative to address these problems. In this strategy, after spinning for the required spin time, the process does not block. Instead, it lowers its priority, boosts the priority of another process (based on the pending messages of the other processes at that workstation), and continues spinning. This avoids an interrupt (since the process keeps spinning albeit at a lower priority), and gives hints to the scheduler as to what should be scheduled next.

In the simulation, the time spent spinning before yielding is again set similar to the SB mechanism. On yielding, the process priority is dropped to a level that is one below the lowest priority of a process at that node and the priority of another process with a pending (unconsumed) message is boosted to the head of level 59 (the details on the algorithm that is used to select the candidate for boosting is described later in the context of the PB mechanism). The application resumes spinning upon returning from the yield mechanism (system call), and the scheduler is likely to preempt this process at the next millisecond boundary. System call costs, together with the overheads for manipulating the priority queues are accounted for in the yield call.

2.4.3 Demand-based Coscheduling (DCS)

Demand-based coscheduling [15] uses an incoming message to schedule the process for which it is intended, and preempts the current process if the intended receiver is not currently scheduled. The underlying rationale is that the receipt of a message denotes the higher likelihood of the sender process of that job being scheduled at the remote workstation at that time.

Our DCS model is similar to the one discussed in [15]. Every 1 millisecond, the NIC finds out which process is currently being scheduled on its host CPU. The NIC uses this information to raise an interrupt (if the receiver process is not currently scheduled) on message arrival after transferring it to the corresponding endpoint. The interrupt service routine simply raises the priority of this receiver process to the head of the queue for level 59 so that it can possibly get scheduled at the next scheduler invocation (millisecond boundary). The application sends and receives (implemented as busy-waits) remain the same as in Local. The model again ensures that the costs for interrupts and queue manipulations are included based on experimental results.

2.4.4 Periodic Boost (PB)

We have proposed this as another interrupt-less alternative to address the inefficiencies arising from scheduling skews between processes. Instead of immediately interrupting the host CPU on message arrival as in DCS, the NIC functionality remains the same as in the baseline ULN receive mechanism. A kernel activity (thread) becomes active every 1 millisecond (the resolution of the scheduler activation), checks message queues and boosts the priority of a process based on some heuristic. Whenever the scheduler becomes active

(at the next millisecond boundary), it would pre-empt the current process and schedule the boosted process.

There are several heuristics that one could use within the PB mechanism to decide on who or when to boost. In this paper, we use one of the eight heuristics identified in [21], which can be explained briefly as follows. The PB mechanism goes about examining message queues in a round-robin fashion starting with the current process, and stops at the first process performing a receive with the message that it is receiving present in the endpoint buffers (message has arrived but has not yet been consumed by a receive call). This process is then boosted to the head of level 59 queue. If there is no such process, then again going about it in a round-robin fashion, the mechanism tries to find a process which is not within a receive call (this can be incorporated easily into the existing ULN mechanism by simply setting a flag in the endpoint when the application enters a receive call, and resetting it when it exits from the call). It then boosts this process if there is one. Else, the PB mechanism does nothing. This algorithm is used in finding a candidate for boosting in the SY mechanism as well.

The simulation models the details of the PB mechanism, incorporating the costs associated with polling the endpoints to find pending message information, and the subsequent costs of manipulating priority queues.

2.4.5 DCS-SB, PB-SB, DCS-SY, PB-SY

One could combine the choices for the two messaging actions as was shown in Table 1 to derive integrated approaches that get the better (or worse) of both choices. As a result, there is nothing preventing us from considering the four alternatives - Demand-based coscheduling with Spin-Block (DCS-SB), Periodic Boost with Spin-Block (PB-SB), Demand-based coscheduling with Spin-Yield (DCS-SY) and Periodic Boost with Spin Yield (PB-SY) - as well.

3. EXPERIMENTAL PLATFORM

Before we present performance results, we give details on the simulation platform, the workloads used to drive the simulator, the parameters that are varied in this exercise, and the performance metrics of interest.

3.1 Simulator

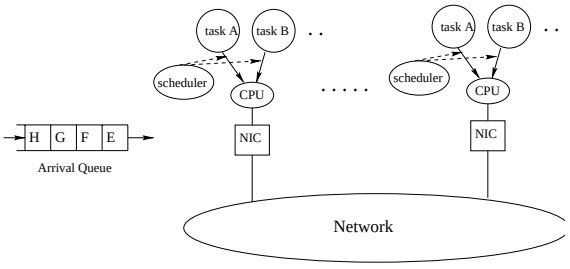


Figure 1: Structure of the Simulator

We use a discrete-event simulator that has been built using the process-based CSIM package. The simulator has the following modules for each node in the system: network interface, operating system scheduler, and the application process. In addition, there is a network module that connects the different nodes. Since the focus of this paper is more on the scheduling mechanisms, we use a simple linear model for the network that is parameterized by the message size and do not consider network contention though the contention at the interface is modeled. The network interface module examines incoming messages from the network and deposits them into the corresponding endpoint. Similarly, it waits for outgoing messages and delivers them into the network module. Costs for these operations have been drawn from microbenchmarks run on our experimental platform discussed in our earlier work [10, 11, 16]. The core scheduler at each node uses a multi-level feedback queue that has been discussed in Section 2.3. The scheduler becomes active every 1 millisecond at each node (similar to the Solaris scheduler). At this time, it checks if the quantum has expired for the currently scheduled process, and if so it preempts and reschedules another. Even if the quantum has not expired, the scheduler consults the feedback queues to check if there is a ready process with a priority higher than the currently scheduled one for possible preemption. There are two other components to the scheduler that correspond to interrupts and the periodic boost mechanism respectively. The interrupt mechanism is used in SB, DCS, PB-SB, DCS-SB, and DCS-SY, and becomes active immediately after the network interface module raises an interrupt. After accounting for interrupt processing costs, the scheduling queues may need to be manipulated in this mechanism. The periodic boost mechanism is used in PB, PB-SB and PB-SY, and becomes active every 1 millisecond to check the endpoints for messages and manipulate the scheduling queues as described in the previous section. Costs for the queue manipulations have again been drawn from our experimental studies.

Our simulator hides all the details of the scheduling models from the application process. The application interface that the simulator offers allows the flexibility of specifying the computation time, communication events (sends or receives with message sizes and destinations), and I/O overheads. The development of the simulator has itself been a significant effort, but we do not delve further into the implementation details.

3.2 Workloads

We are interested in using realistic workloads to drive the performance evaluation. Towards this, we are interested in capturing the dynamic behavior of the environment (i.e. dynamic job arrivals), different job execution times and job sizes (number of CPUs), and the characteristics of each job (computation, communication and I/O fractions) as well.

To capture the dynamic behavior of the environment, our experiments use a workload that is drawn from a characterization of a real supercomputing environment at Lawrence Livermore National Labs. Job arrival, execution time and size (number of CPUs - henceforth referred to as tasks) information of this environment have been traced and characterized to fit a mathematical model (Hyper-Erlang distribution of common order). The reader is referred to [6] for details on this work and the use of the model in different evaluation exercises [20]. However, due to the immense simulation details involved in this exercise (unlike in any of the previous studies) requiring the modeling of scheduling queues at

granularities smaller than even a millisecond, it is not feasible to simulate very large systems. As a compromise, we have limited ourselves to clusters of upto 32 nodes, and select jobs from the characterized model that fall within this limit. However, we feel that the general trends and conclusions would apply to larger systems as well.

As can be expected, the characteristics of each job in the system can further have an impact on the performance results. In particular, the time spent in the computation (only the CPU is required), communication and I/O activities, and the frequency of these operations, can interact with the scheduling strategies in different ways. It is easy to draw false conclusions if one does not consider all these different artifacts in the performance evaluation. In reality, a job can be intensive in any one of the three components - computation (CPU), communication or I/O - or can have different proportions of these components. To consider these different situations, we identify six different job types with different proportions of these components. Our evaluations use eight different workloads, termed *WL1* to *WL8*, with the first six using jobs of the corresponding class in isolation as shown in Table 2. In the seventh workload (WL7) a job has an equal probability of falling in any of the six job types, so that we consider the effect of a mixed load on the system. The last workload (WL8) considers an equal mix of the three job classes that are each intensive in one of the three components (CPU, I/O and communication).

Job Type	Comp. (%)	I/O (%)	Comm. (%)
J1	35	15	50
J2	35	50	15
J3	35	35	30
J4	90	5	5
J5	35	5	60
J6	65	5	30

Workload	Job Types in Workload
WL1	J1
WL2	J2
WL3	J3
WL4	J4
WL5	J5
WL6	J6
WL7	equal mix of J1 thru' J6
WL8	equal mix of J2, J4 & J5

Table 2: Workloads

For the results presented in this paper, the communication is nearest-neighbor (i.e. Task i of a job talks to Tasks $i-1$ and $i+1$) with a constant message size of 4096 bytes. We have experimented with other communication parameters/patterns as well, and we find that the overall results/trends presented in this paper still hold. Once the relative proportions and corresponding times in the three components are derived for a given job, its tasks iteratively go through a sequence of compute, I/O, sends/receives to/from its nearest neighbors as shown in Figure 2. By fixing the raw 1-way latency of a 4096 byte message (from an experimental platform), the cost of communication per iteration in the ideal case (when everything is balanced) is known. Based on this and the relative proportion of the other two components (compute and I/O) as determined by the job type, the computation and I/O times per iteration can be calculated. Together with the total job execution time (given by the characterized model), these individual times determine the number of iterations that a job goes through.

It is also important to note that the work imbalance/skewness (and the resulting mismatch of the sends and receives) can have a significant impact on results. As a result, the skewness (s), which is expressed as a percentage of the computation and I/O fractions of the tasks, is another parameter that is varied in the experiments. Formally, the CPU and I/O components of each iteration of a task that were calculated earlier, are each multiplied by a factor $(1+unif(-s/2, s/2))$ where $unif(x, y)$ generates a random number between x and y using a uniform distribution. If s is set to 0, then all tasks spend the same computation and I/O times in each iteration, and thus arrive at the communication events at the same time. In this case, the execution time for this job will match the one picked for it from the characterized model [6] on a dedicated (non-multiprogrammed) system. A larger skewness implies that tasks will arrive at the communication events at different times, and the overall execution time per iteration will depend on who comes last to the send/receive calls. This also implies that the execution time is likely to get larger compared to that derived from the characterized model (each iteration can get elongated) with a larger s. The effectiveness of the scheduling mechanisms can be evaluated by how well they are able to hide the increase in execution times.

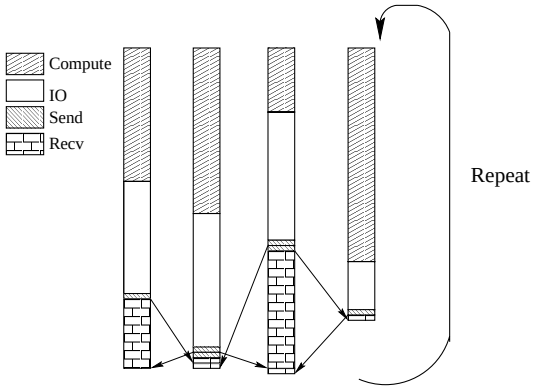


Figure 2: Job Structure

3.3 Parameters

Several parameters and costs can be varied in the simulator, and some of these (and the values that are used) are given in Table 3. Many of the times given in this table have been obtained from microbenchmarks on actual operating systems and Myrinet clusters with a ULN connecting the machines [10, 16].

3.4 Metrics

This exercise considers several metrics that are important from both the system and user's perspective:

- *Response Time:* This is the time difference between when a job completes and when it arrives in the system, averaged over all jobs.

Parameter	Value(s)
p (# of nodes or CPUs)	32, 16
MPL (Multiprogramming Level)	2, 4, 5, 16
s (Skewness)	150%, 20%
CS (Context Switch Cost)	
Dynamic Coscheduling	200 us, 100 us
GS	2 ms, 1 ms
I (Interrupt Cost)	50 us, 25 us
Q (GS Time Quantum)	200 ms, 100 ms
Move between queues	3 us
Check an endpoint	2 us
Message Size	4096 bytes
1-way Message Latency	185.48 us

Table 3: Simulation parameters and values used in experiments. Unless explicitly mentioned otherwise the default (underlined) values are used.

- *Wait Time:* This is the average time spent by a job waiting in the arrival queue before it is scheduled.
- *Execution Time:* This is the difference between *Response* and *Wait* times.
- *Throughput:* This is the number of jobs completed per unit time.
- *Utilization:* This is the percentage of time that the system actually spends in useful work.

We have also examined fairness issues, but the results are not presented here due to space limitations. The reader is referred to [21] for these results and detailed performance profiles of the executions.

4. PERFORMANCE RESULTS

Even though we have conducted numerous experiments varying the different parameters to obtain the different performance metrics for all workloads, we present only representative results.

4.1 Impact of Load

As the load increases (higher arrival rates and job execution times), the system is likely to be more heavily utilized. As a result, jobs are likely to experience longer wait and response times. The job characterization effort [6] provides a way of cranking the induced load, and the resulting effect on the response time of the system is plotted against the system utilization in Figure 3 for the mixed workload (WL7).

Local, SY, DCS and DCS-SY, which all use spin-based receives, saturate even before the utilization reaches 50%, and are thus not seen in the Figure. We find that GS can go only as high as 57% before saturation. The remaining schemes - DCS-SB, PB-SY, PB-SB, SB and PB - perform significantly better than GS. Of these, PB, PB-SB and SB, perform the best with the utilization going as high as 77% before saturation.

4.2 Impact of the nature of workload

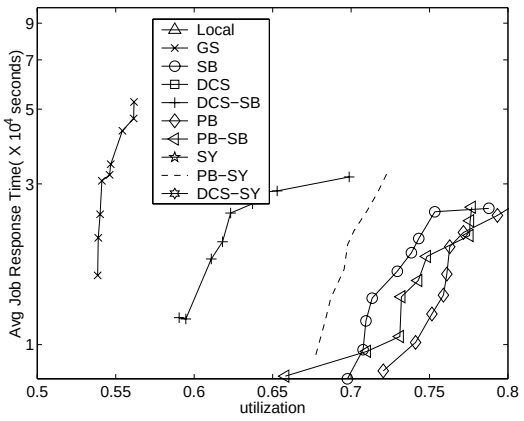
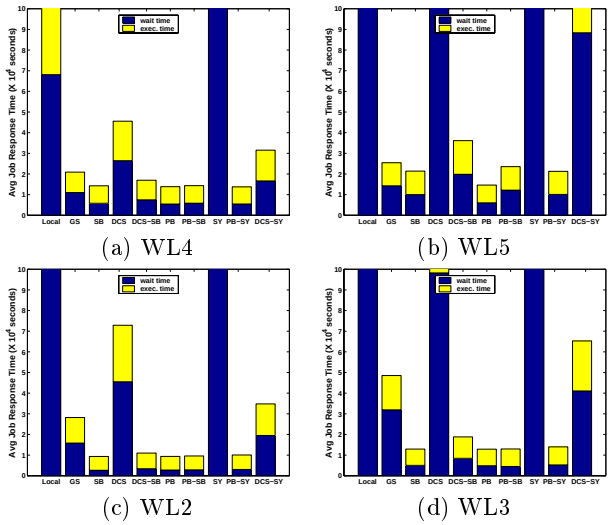


Figure 3: Impact of Load on Response Time (WL7, p=32, MPL=5, s=20%)

Next, we examine the impact of the computation, communication and I/O components of the workload on the performance of the different schemes. WL4, WL5, WL2 and WL3 are CPU intensive, communication intensive, I/O intensive and evenly balanced (between the three components) workloads respectively. The response times of the ten alternatives for these four workloads are presented in Figure 4. The response time is further broken down into the wait time in the arrival queue, and the execution time. It should be noted that some of the bars which hit the upper boundary of the y-axis have been truncated (and the execution portion of these bars is not visible).



(a) WL4 (b) WL5

(c) WL2 (d) WL3

Figure 4: Impact of the nature of Workload on Response Time (p=32, MPL=5, s=20%)

For the CPU intensive workload (WL4), Local and SY are simply not acceptable (this is true for the other workloads as well). Both DCS and DCS-SY have higher response times than the rest, mainly because of the longer wait in the arrival queue. GS has the next highest response time for this workload. For a high computation proportion, even a 20% skewness can make a difference in causing a load imbalance between the tasks, leading to inefficiencies in GS. There is

not a significant difference between the response times of the other five schemes for this workload. When the computation is high, the inefficiencies due to in-exact coscheduling is not really felt (since receives are infrequent), and any skewness is well hidden by the scheduling mechanism.

On the other hand, when one examines the communication intensive workload (WL5), the differences are more pronounced. DCS and DCS-SY become much worse here. The performance of GS, SB, PB-SB and PB-SY is comparable, with PB giving the best performance. When the communication component becomes high, there is a stronger need for coscheduling. Further, the skewness between the executing tasks of an application gets lower with a lower compute fraction, making the inefficiencies of GS less important.

On the other hand, in the I/O intensive (WL2) or mixed workload (WL3), the occurrence of I/O activities within a time quantum keeps the CPU idle for the remainder of the quantum in GS. When the I/O portion is more intensive (WL2), the impact of non-coscheduled execution is not felt. As a result, there is not a significant difference between SB, DCS-SB, PB, PB-SB and PB-SY. Finally, the evenly balanced workload (WL3) reiterates the observations made for the communication intensive workload, though to a lesser degree, while showing that GS is not as good as SB/PB/PB-SB/PB-SY because of the presence of I/O activities.

As in the previous subsection, we find that the PB, SB, PB-SY and PB-SB schemes are uniformly good for all the workloads. Of these, PB is clearly the best for the communication intensive workload. DCS-SB falls in the next category. GS is reasonable as long as the execution does not lead to work imbalances.

4.3 Impact of Multiprogramming level (MPL)

As one increases the multiprogramming level, larger number of jobs can be simultaneously accommodated in the system which works in favor of lowering wait times. Larger MPL also allows the system to find alternate work (that is useful) to do when processes block (or yield). However, larger MPLs also imply a larger number of context switches during execution, which increases execution times. In addition, larger MPL also decreases the likelihood of the tasks being coscheduled in the dynamic coscheduling mechanisms. It is thus interesting to study the impact of MPL to understand these factors and their interplay. Table 4 shows the change in response times normalized (with respect to MPL=2) for WL2 with MPL=5, and WL4 and WL5 with MPL=5 and 16. The reader is referred to [21] for the detailed performance profile graphs of these executions.

In Local, the overheads with a larger MPL dominate over other factors resulting in larger response times when we go from MPL level of 2 to 5 for all 3 workloads. Moving from MPL level of 2 to 5 results in lowering the percentage of time spent in useful computation (increasing the spin component in the spinning alternatives). In these experiments again, we find that schemes incorporating PB and SB in some form or the other, are able to provide a more scalable improvement in response times with increasing MPLs compared to the rest. At lower MPLs, the SB mechanisms are not able to keep the CPU fully occupied. (reader is referred to [21] for the idle times in the performance profile). This is even more significant for the I/O intensive and communication intensive workloads which block more frequently. When the MPL is increased, while the context switch times do go up, it is seen that the reduction in the idling is more than adequate to compensate for any overheads. With the PB mechanism, we find that it is better than SB at lower MPL levels (because it blocks less frequently), particularly for the communication intensive workload (WL5). SB really needs much higher MPL levels before its performance becomes comparable to PB. From the workload viewpoint, I/O and communication intensive workloads have larger changes (and will better benefit) with MPL compared to the CPU intensive workload.

Scheme	WL2	WL4		WL5	
	MPL=5	MPL=5	MPL=16	MPL=5	MPL=16
Local	0.30	0.37	-	15.44	-
GS	-0.25	-0.20	-0.34	-0.30	-0.41
SB	-0.69	-0.36	-0.48	-0.44	-0.54
DCS	-0.27	-0.13	-0.34	0.18	-
DCS-SB	-0.68	-0.35	-0.44	-0.21	-0.38
PB	-0.65	-0.25	-0.38	-0.41	-0.52
PB-SB	-0.71	-0.34	-0.45	-0.43	-0.54
SY	-0.29	2.82	-	4.59	-
PB-SY	-0.64	-0.24	0.36	-0.28	2.78
DCS-SY	-0.58	-0.44	-0.81	-0.42	-0.94

Table 4: Impact of MPL: Normalized Change in Response Time with respect to MPL = 2 (p=32, s=20%)

It is to be noted that after a point, one can expect response times to eventually go up (even though this is shown for only 1 of the schemes at MPL = 16 in Table 4) due to the overheads dominating any potential benefits.

4.4 Impact of Skewness

Skewness (work imbalance) between the tasks of a job can determine the amount of time that a receiver spins or blocks for a message. Hence, it would be interesting to study how the mis-match of the sends and receives affects the performance of the scheduling mechanisms. Figure 5 shows the response times for the different schemes with two different skewness values (20% and 150%) for the CPU intensive workload.

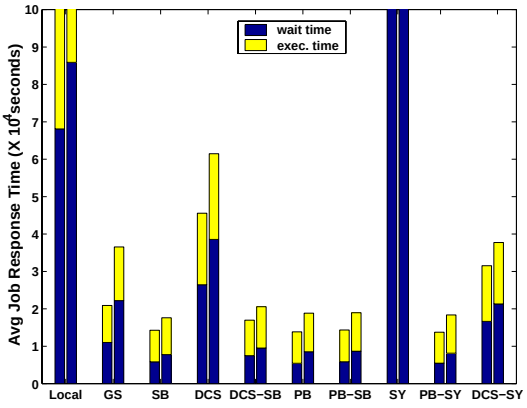


Figure 5: Impact of Skewness on Response Time (WL4, p=32, MPL=5, s=20% (left bar) and 150% (right bar))

Even if a job is running in isolation on a dedicated system, increasing the skewness would increase its execution time (as per the explanation for skewness given in Section 3.2). This effect is clearly seen for GS on the two workloads where the execution time goes up from s=20% to 150% since GS does not have much scope for hiding the impact of this skewness. On the other hand, the dynamic coscheduling mechanisms can potentially better utilize the CPUs than GS for larger skewness values. We can see that most of these schemes are better able to hide the effect of the skewness compared to GS; the spin time increase for a larger skewness for the spinning mechanisms, and the idle time increase for the blocking mechanisms, are not as high as the increase in spin times for GS (such performance profiles are not given here due to space limitations and the reader is referred to [21] for further details). Between the spinning and blocking mechanisms, we find that the latter are better able to hide the effect of skewness as can be expected. We have also tried a similar experiment for the communication intensive workload, and we have found that the effect of skewness is less pronounced for this workload. This is because skewness is varied as a percentage, rather than as an absolute value. As a result, the skew is higher in the compute intensive workload than in the communication intensive workload. The latter tends to inherently synchronize the tasks more often.

4.5 Impact of System Overheads

In all of the above discussion, the context switch and interrupt processing costs were kept constant at 200 and 50 microseconds respectively for all schemes except GS (where the context switch was set at 2 milliseconds to account for the synchronization between the CPUs). These parameters can also have an effect on the relative performance of the schemes and Table 5 shows these effects for SB, DCS, DCS-SB, PB, PB-SB and GS. The response times in this Table have been normalized with respect to the first column. It should be noted that there are no interrupts in the PB and GS mechanisms.

Scheme	$Q = 200ms$, $CS = 2ms$	$Q = 100ms$, $CS = 1ms$	$Q = 100ms$, $CS = 2ms$
GS	1.000	0.896	0.977
	$CS = 200us$, $I = 50us$	$CS = 100us$, $I = 25us$	$CS = 100us$, $I = 50us$
SB	1.000	0.647	0.659
DCS	1.000	0.902	0.904
DCS-SB	1.000	0.710	0.804
PB-SB	1.000	0.670	0.687
PB	1.000	0.89	

Table 5: Impact of System Overheads on Response Time (WL5, p=32, MPL=5, s=20%)

The context switch times in GS are determined not only by the costs of swapping in/out processes at each node, but also due to the explicit synchronization between the nodes. As a result context switch costs in GS are usually much higher than for the schemes which do not require any explicit synchronization. If we keep the ratio of the time quantum to the context switch overhead the same (100:1), we find that a smaller quantum helps this scheme. This is because with a smaller quantum, the amount of time in the quantum that is wasted due to blocking (for I/O) or spinning (for a message) becomes smaller. By the time the process gets rescheduled, the operation may be complete, thus allowing better overlap with useful computation. As is to be expected, increasing the overhead percentage due to context switches (i.e. column 3 compared to column 2), extends the execution time (and thereby the response time) of a job.

With the overheads dropping (column 2 compared to column 1), the response times reduce for the different coscheduling heuristics. We find the benefits more significant for the blocking mechanisms compared to those that employ spinning. Blocking executions typically involve many more context switches (each block involves a switch), and thus benefits from the lower associated costs. As interrupt costs go up with no changes to the context switch costs (column 2 to 3), the response times for the mechanisms which incur interrupts (DCS, DCS-SB, SB and PB-SB) increase.

5. CONCLUDING REMARKS

Advances in user-level networking (ULN) allows us to explore a new domain of dynamic coscheduling mechanisms that offer a potential improvement over conventional scheduling strategies (such as space sharing or coscheduling) for parallel machines. These mechanisms become even more important for cluster environments where applications with diverse characteristics and QoS requirements coexist. Until now, the understanding and knowledge of the relative performance of dynamic coscheduling mechanisms is rather limited [10, 3, 1, 15]. While our previous work [10] did a preliminary examination of these dynamic coscheduling mechanisms, the study was rather limited due to the workloads that were considered, the experiments that were conducted, the inflexibility of the underlying system in modulating parameters, and the performance metrics that were evaluated. For the very first time, this paper has presented a comprehensive evaluation study of the different dynamic coscheduling alternatives using realistic and dynamic workloads with varying job sizes, execution times, and characteristics, in studying the impact of different system and workload parameters on numerous performance metrics. This exercise has required the development of a comprehensive and flexible simulator, which has itself been a substantial effort. Using this simulator, we have been able to examine the suitability of different scheduling mechanisms under varying conditions.

There is clearly a great need for some coordinated scheduling effort between the nodes to accommodate the parallel jobs. Gang scheduling (GS), as has been the norm, results in high wait times in the arrival queue. Further, GS executions do not fare as well as the dynamic coscheduling alternatives when: (a) the skewness between the tasks of a job is high (GS is not able to hide the spin times as well as the dynamic coscheduling mechanisms); (b) the jobs are I/O intensive (GS wastes the rest of the time quantum); and (c) when the costs for explicitly synchronizing the nodes between time quanta becomes high (perhaps, for large systems). Of the dynamic coscheduling mechanisms, we find Periodic Boost (PB) outperforming the other schemes. It is able to get the advantages of spinning (to avoid interrupt processing costs), and is able to relinquish the CPU whenever needed.

We have not been able to delve into the issue of different

heuristics for the Periodic Boost mechanism and their fairness in this paper due to space limitations. The reader is referred to [21] for a preliminary investigation into this area, and we plan to explore this issue in greater depth in the future. We are also interested in theoretically modeling the scheduling mechanisms/heuristics so that we can understand how well they perform (and how close they are to the optimal). Recently, a related study [14] shows how buffering and latency tolerance (separating the posting of a receive from the time when the receiver should actually block) can be used to minimize the impact of non-coscheduled execution. However, their study assumes that all the communication latency can be hidden, in which case coscheduling is not important. However, this is not completely realistic, and we propose to explore the possibility of exploiting communication slackness (gap between posting of a receive and the time the data is actually needed) within the PB heuristics for even better performance.

6. REFERENCES

[1] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the ACM SIGMETRICS 1998 Conference on Measurement and Modeling of Computer Systems*, 1998.

[2] M. Buchanan and A. Chien. Coordinated Thread Scheduling for Workstation Clusters under Windows NT. In *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[3] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the ACM SIGMETRICS 1996 Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 1996.

[4] D. G. Feitelson and L. Rudolph. Coscheduling based on Run-Time Identification of Activity Working Sets. Technical Report Research Report RC 18416(80519), IBM T. J. Watson Research Center, October 1992.

[5] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.

[6] H. Franke, J. Jann, J. E. Moreira, P. Pattnaik, and M. A. Jette. Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. In *Proceedings of Supercomputing*, November 1999.

[7] A. Hori, H. Tezuka, and Y. Ishikawa. Global State Detection Using Network Preemption. In *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pages 262–276, April 1997. LNCS 1291.

[8] D. Lifka. The ANL/IBM SP Scheduling System. In *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, April 1995. LNCS 949.

[9] J. E. Moreira, H. Franke, W. Chan, L. L. Fong, M. A. Jette, and A. Yoo. A Gang-Scheduling System for ASCI Blue-Pacific. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking(HPCN'99), volume 1593 of Lecture Notes in Computer Science*, pages 831–840, April 1999.

[10] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. A Closer Look at Coscheduling Approaches for a Network of Workstations. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 96–105, June 1999.

[11] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. Alternatives to Coscheduling a Network of Workstations. *Journal of Parallel and Distributed Computing*, 59(2):302–327, November 1999.

[12] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[13] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, December 1995.

[14] F. Petrini and W. Feng. Buffered Coscheduling: A New Method for Multitasking Parallel Jobs on Distributed Systems. Technical report, Los Alamos National Laboratory, September 1999.

[15] P. G. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, January 1997.

[16] R. Subrahmaniam. Implementing Coscheduling Heuristics for Windows NT Clusters. Master's thesis, Dept. of Computer Science and Engineering, Penn State University, University Park, PA 16802, October 1999.

[17] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, October 1991.

[18] Specification for the Virtual Interface Architecture. `http://www.viarch.org`.

[19] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[20] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. In *Proceedings of the International Parallel and Distributed Processing Symposium*, May 2000. To appear.

[21] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke. A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment. Technical Report CSE-99-022, Dept. of Computer Science and Engineering, The Pennsylvania State University, November 1999.