Impact of Workload and System Parameters on Next Generation Cluster Scheduling Mechanisms *†

Yanyong Zhang *Anand Sivasubramaniam* *José 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, unattractive. 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, offers 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. In addition, heuristics to enhance one of the alternatives even further are identified, classified and evaluated. It is shown that these heuristics can significantly outperform the other alternatives over a spectrum of workload and system parameters, and is thus a much better option for clusters than conventional coscheduling.

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.

†This is a substantially revised and extended version of a preliminary paper [50] that appeared in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 100-109, May 2000. The revisions include a more detailed evaluation, consideration of more workload and system parameters, and the design and evaluation of a novel set of heuristics to be used in conjunction with the proposed Periodic Boost mechanism.

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 [12, 14], 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 can raise complications when 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).

With the growing popularity of clusters, there have been several academic and commercial endeavors to develop networks, network interfaces, and messaging layers to provide efficient communication support for these systems. However, there is the more important and complex problem of managing and coordinating the resources across the nodes of a cluster. Optimizing communication alone may not necessarily translate to improved performance since a poor scheduling strategy could nullify any savings. For instance, a currently scheduled process on one node would experience a long wait for a message from a process not currently scheduled on another node regardless of the low network latency. Scheduling and communication issues are thus closely intertwined, and should be studied together. An efficient scheduling solution should understand the communication and I/O characteristics of the processes, and coordinate the different activities across the nodes of a cluster, without adding significant overheads.

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 [34, 18, 47, 41, 25, 33, 35, 6, 46, 48] 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 [19, 42], 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 is efficient from the viewpoint of reducing context switching costs, as well as not having to hold the working sets of multiple applications in memory. 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 [26, 10, 9, 40, 11, 45], that combines time sharing with space sharing, to remedy some of these problems. 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 or waiting for a message within its allotted time quantum [11].

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. Specifically, several user-processes at a node could concurrently send a message, and an incoming message is directly transferred by the NIC to the corresponding destination process at that node (even if that process is not currently scheduled). It is thus not necessary to perform network context switching when the processes are switched out on their respective CPUs as was necessary [40, 13] 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 [23, 24], there were only two suggestions [1, 37, 4] 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 an experimental cluster of eight Sun Ultra Enterprise servers running MPI applications [23, 24]. 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, number of CPUs, and the operating system costs for context switching and interrupt processing affect the relative performance of the schemes?
- Within the purview of the flexibility offered by the Periodic Boost mechanism that we have proposed, what coscheduling heuristics can we employ, and how do these heuristics fare?

Investigation of these issues requires an extensive framework to model and evaluate the different scheduling alternatives. We have developed this framework with a detailed simulator that uses a multi-level feedback queue model (based on the Sun Solaris scheduler) as a core, and builds the different scheduling alternatives around it. The choice of a simulation-based study stems from the need to vary different system parameters (some of which are difficult or even impossible to tweak on an actual system) in a controlled/non-intrusive manner. The simulator is driven by an abstraction of a real workload [11] 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 (response time, wait time, utilization, and fairness) and dynamic workloads. In addition, this paper identifies several new heuristics that can be used in conjunction with the Periodic Boost mechanism. Some of these heuristics can significantly outperform the other alternatives in terms of response time, utilization and fairness, and are better equipped to handle the responsiveness demands of emerging applications.

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 since it is important for the rest of the discussion. 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 [23, 24] 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 memory, and receiving translates to (waiting and) dequeuing a message from memory. 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, while most traditional kernel-based mechanisms are an order of magnitude more expensive. User-level messaging is achieved without compromising protection since each process can only access its own send/receive buffers (referred to as an *end-point*). Thus, virtual memory automatically provides protected access to the network. Several ULNs [43, 44, 27, 2] based on variations of this paradigm have been developed.

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 CM-5 [40], Meiko CS-2 (gang mode) [3], Intel Paragon (gang mode) [15], AP/Linux [39] and the GangLL [11] on the IBM SP-2 at LLNL, are examples of systems that use this approach.

The simulation model is based on the implementation of the GangLL scheduler [22, 11] on the Blue Pacific machine at Lawrence Livermore National Labs. The model uses an Ousterhout [26] 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 [11] 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/Heuristics

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 mechanism. 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 our earlier work [23]. 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 [8, 1, 30] and demand-based coscheduling [37]. 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 to be 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 [23], 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 [37] 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 [37]. 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, and in this paper, *we propose, classify and evaluate eight new heuristics*. For most of the experiments, the heuristic that is used (called Heuristic **A**) can be briefly explained as follows (further details are in Section 4.9). 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 the 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 rationale behind this heuristic, and a comparison with seven other variations is given later in Section 4.9. 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 Combinations (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 best (or worst) 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 workload used to drive the simulator, the parameters that are varied in this exercise, and the performance metrics of interest.

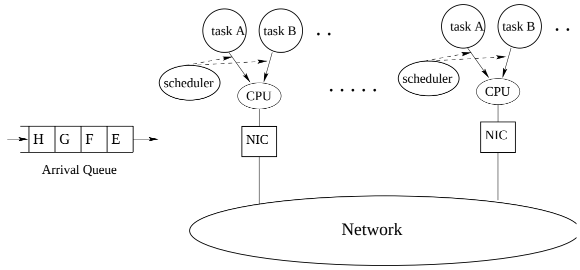


Figure 1: Structure of the Simulator

3.1 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 network 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 [23, 24, 38]. 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 re-schedules 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 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 [11] for details on this work and the use of the model in different evaluation exercises [49]. 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

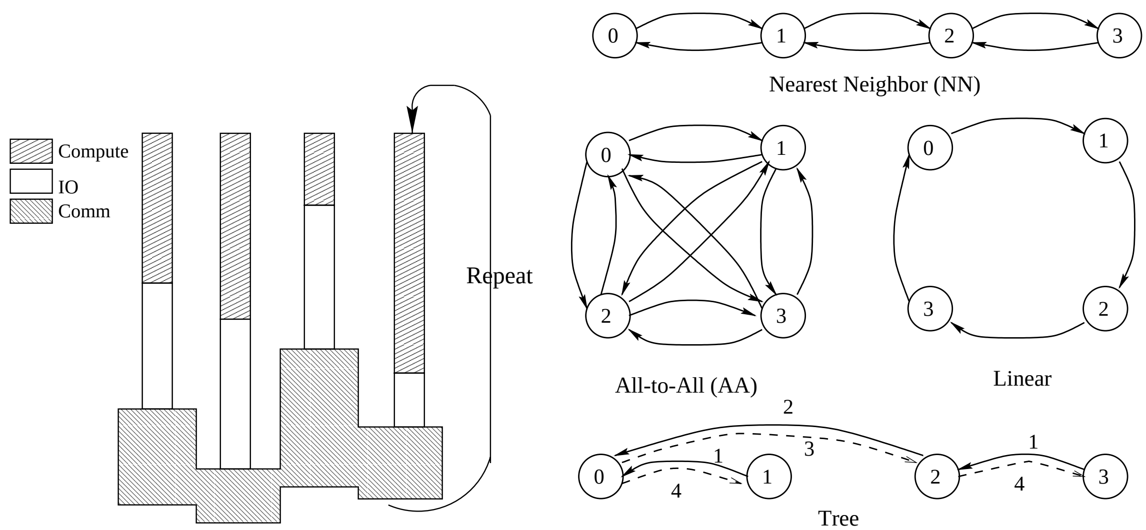
Workload	Job Types in Workload
WL1	J1
WL2	J2
WL3	J3
WL4	J4
WL5	J5
WL6	J6
WL7	equal mix of J1 thru' J6
WL8	equal mix of J2, J4 & J5

Table 2: Workloads

We consider four different communication patterns (that are typical of several parallel applications) between the tasks of a job that are shown in Figure 2(b). The considered patterns include *Nearest Neighbor (NN)* (process i communicates with $i-1$ and $i+1$), *all-to-all (AA)* (process i sends a message to all other processes and waits for a message from everyone else, which is typical in many scientific computations such as matrix transposes, multidimensional FFTs etc.), *tree* (going up and coming down, similar to the tree-based barrier discussed in [21], which is a typical structure used for global reductions), and *linear* (process i sends a message to $i+1$ and waits for a message from $i-1$, which is a common structure in pipelined executions). All these patterns use a constant message size of 4096 bytes.

Once the relative proportions and corresponding times in the three components (computation, communication and I/O) are derived for a given job, its tasks iteratively (as shown in Figure 2 (a)) go through a sequence of compute, I/O, sends/receives as per the communication pattern (Figure 2)(b). 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) which is 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 between the tasks (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 [11] 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.



(a) application model (b) communication pattern

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 [23, 38].

3.4 Metrics

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

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 GS	 200 us, 100 us 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
Communication Pattern	NN, AA, Tree, Linear
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.

- *Response Time:* This is the time difference between when a job completes and when it arrives in the system, averaged over all jobs.
- *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.
- *Slowdown:* This is the ratio of the *execution time* to the time taken on a system dedicated solely to this job. It is an indication of the slowdown that a job experiences when it executes in multiprogrammed fashion compared to running in isolation, after it is allocated to the CPUs.
- *Utilization:* This is the percentage of time that the system actually spends in useful work.
- *Fairness:* The fairness to different job types (CPU, communication or I/O intensive) is evaluated by comparing (the coefficient of variation of) the slowdown of the individual job classes in a mixed workload. A smaller variation indicates a more fair scheme.
- *Performance Profile:* This is expressed as a graph showing the percentage of time that a CPU spends on the average in different components (compute, spinning/busy-wait, context switches, idling, and other overheads such as interrupt costs). Such a graph is useful to better understand the performance results, and also to identify bottlenecks in the execution.

4 Performance Results

4.1 Impact of Load

As the load increases, the system is likely to be more heavily utilized. Consequently, jobs are likely to experience longer wait and response times. The workload characterization effort presented in prior research [11] provides a way of cranking the induced load by modulating the job arrival rates and execution times. The resulting effect on the response time of the system is plotted against the system utilization in Figure 3 for the mixed workload (WL7).

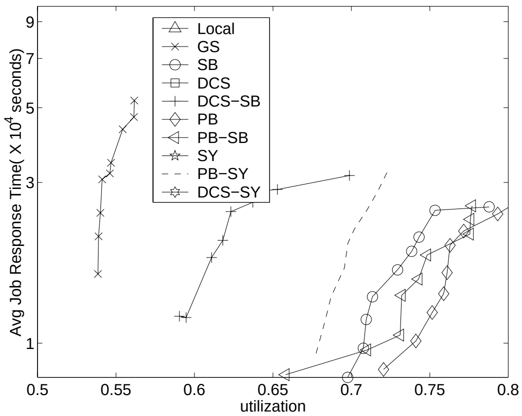


Figure 3: Impact of Load on Response Time (NN, WL7, p=32, MPL=5, s=20%)

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 Figure 3. 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 reaching as high as 77% before saturation.

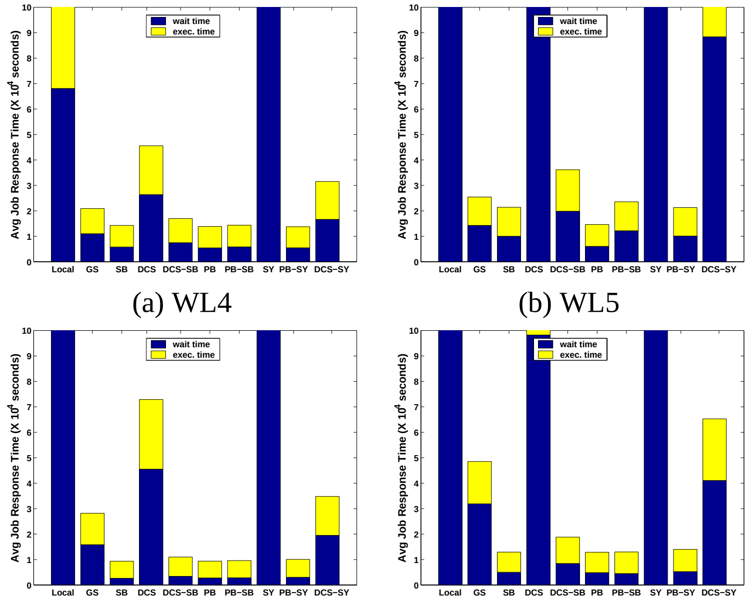
4.2 Impact of the nature of workload

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 scheduling 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).

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 s=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.

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



(a) WL4 (b) WL5

(c) WL2 (d) WL3

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

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. It should be noted that in all four workloads, several of the dynamic coscheduling mechanisms have a shorter wait time component compared to GS. This is because allocating CPUs to a job in GS requires those CPUs to be available during the same time quantum (i.e. the same row of the Ousterhout matrix), while the others do not pose any such restrictions.

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 stretches 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. Figure 5 shows the performance profile graphs for the corresponding experiments, which can be used to observe the change in percentage of time spent in different activities when one increases the MPL level.

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. We can glean this behavior from the performance profile graphs as well. 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

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 (NN, p=32, s=20%)

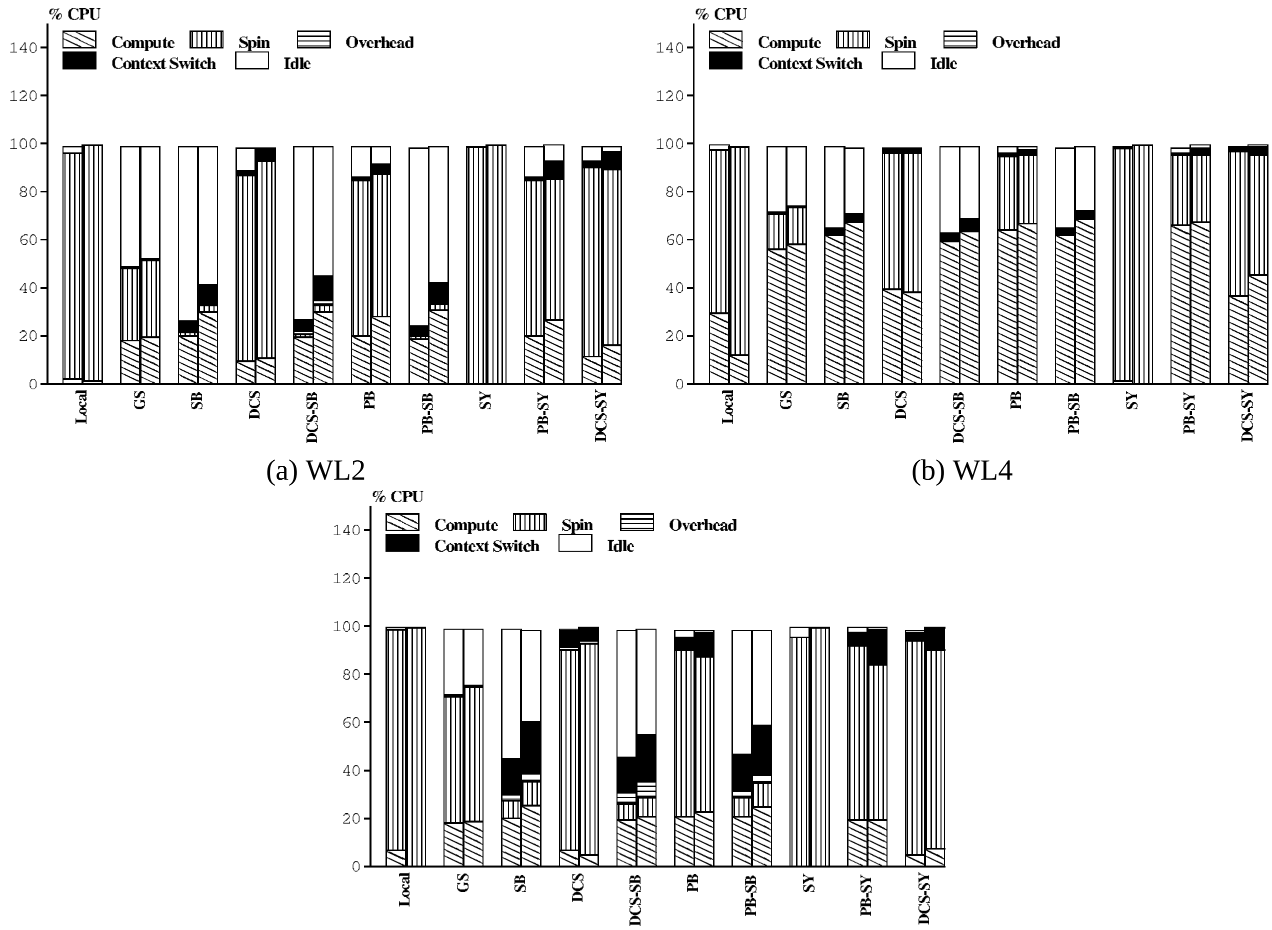
in response times with increasing MPLs compared to the rest. At lower MPLs, the SB mechanisms are not able to keep the CPU fully occupied (idle time in the performance profile). This is even more significant for the I/O intensive and communication intensive workloads which block more frequently. When 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.

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 6 shows the response times and corresponding performance profile for the different schemes with two different skewness values (20% and 150%) for the CPU intensive workload.

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 with 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 (as seen in performance profiles). 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



(a) WL2 (b) WL4

(c) WL5

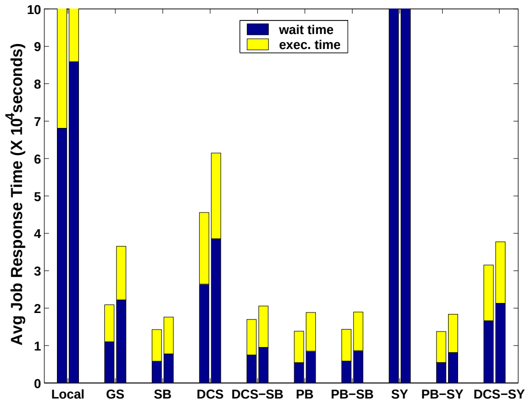
Figure 5: Impact of MPL: Performance Profile (NN, p=32, MPL=2 (left bar) and 5 (right bar), s=20%)

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 inherently synchronizes the tasks more often.

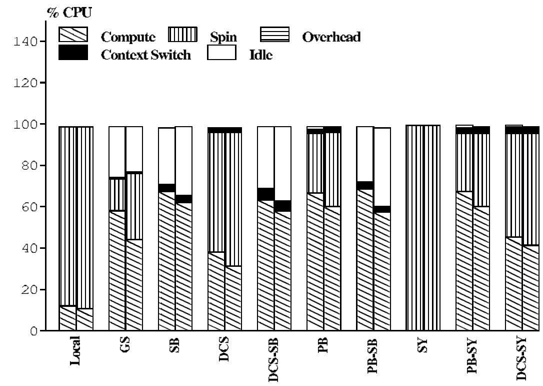
4.5 Impact of System Overheads

In the above discussions, 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.

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



(a) Response Time



(b) Performance Profile

Figure 6: Impact of Skewness (NN, WL4, p=32, MPL=5, s=20% (left bar) and 150% (right bar))

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	

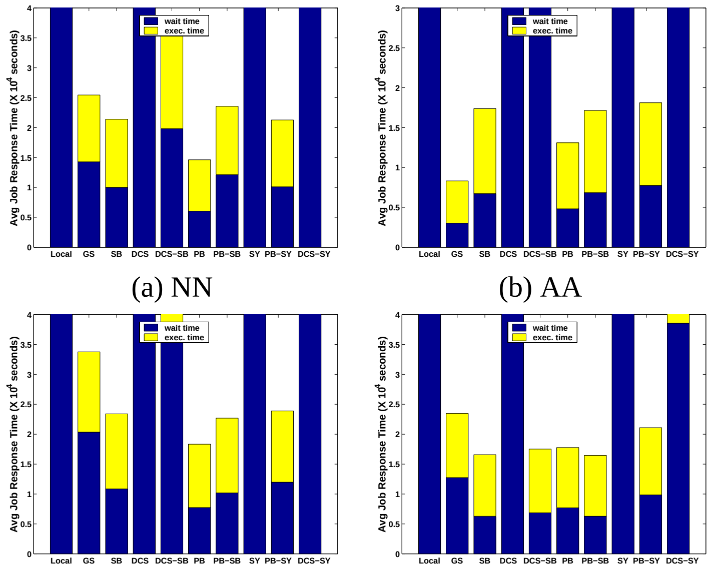
Table 5: Impact of System Overheads on Response Time (NN, WL5, p=32, MPL=5, s=20%)

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 can thus benefit 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. PB makes rescheduling decisions at a much coarser granularity, and is thus less likely to be affected by system overheads.

4.6 Impact of Communication Pattern

Figure 7 (a), (b), (c) and (d) shows the response times with the different schemes for the four communication patterns that have been considered. Examining the tree pattern, we find that the blocking schemes perform better than the other alternatives, while this is not necessarily the case in the other patterns. With this pattern, sends and the corresponding receives at the other end are more likely to be mismatched (i.e. receive may happen much before the send). This is similar to what we observed with a higher skewness factor, making the blocking schemes perform rather well. With the all-to-all pattern, coscheduling becomes extremely important, and one cannot tolerate any mismatch of sends and receives. As a result, we find the GS scheme performing the best. Linear and nearest-neighbor show comparable behavior for the different schemes, and these patterns fall between the other two (tree and all-to-all) in terms of how important it is to coschedule the processes. Regardless of the communication pattern, we find the PB mechanism uniformly giving good performance.



(a) NN (b) AA

(c) Linear (d) Tree

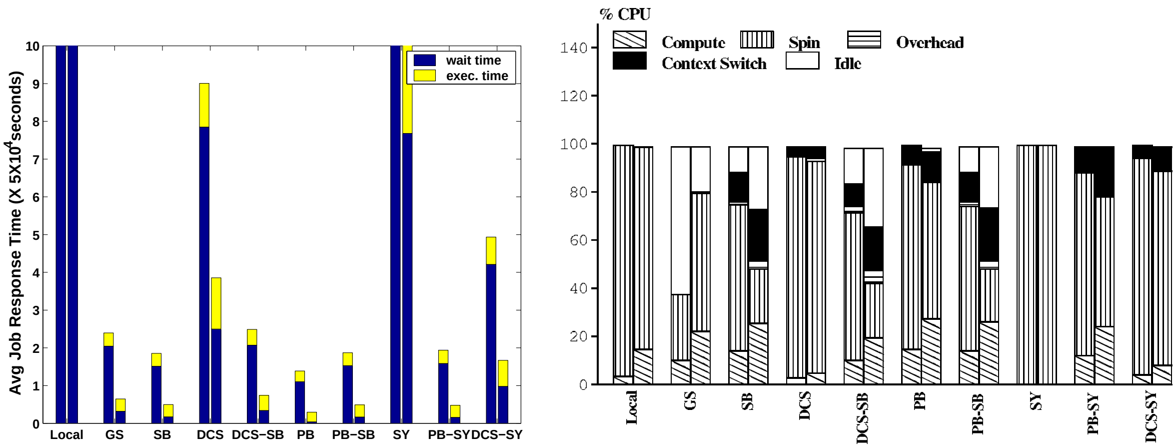
Figure 7: Impact of the nature of Communication Pattern on Response Time (p=32, MPL=5, WL5, s=20%)

4.7 Impact of System Size

It is interesting to find out what would be the improvement in response time as we increase the number of CPUs in the system (scalability issue). It is clear that the wait time in the arrival queue would go down. However, increasing system size (and keeping the same number of processes at each node) lessens the likelihood of coscheduled execution due to the higher possibility of conflicting decisions being made across the nodes (stretching the execution time). The relative changes in these two components will determine the scalability of the system. To evaluate these issues, we conduct experiments with two system sizes (16 and 32 CPUs), and use the same load (with each job demanding at most 16 CPUs) on the two configurations. The resulting response times are shown in Figure 8 together with its performance profile.

We observe that doubling the system size significantly (more than halving) lowers the wait time (which is the dominating factor of the response time) in the arrival queue. This more than adequately compensates for the minor increases in execution time (due to lower probability of coscheduled execution). Though the results are not explicitly shown here, it should be noted that these factors are accentuated with higher loads, where the wait times are likely to be even higher.

The performance profile graph also sheds interesting insights on scalability of the system. For the blocking schemes (SB, DCS-SB and PB-SB), we find that the idle time component of the execution goes up. This is because the same load is exercising a larger system, resulting in lower load (tasks assigned or effective MPL) per node. Lower effective MPLs imply a larger idle time for the blocking mechanisms (since they run out of work), and a lower spin component in the spinning mechanisms due to the higher likelihood of coscheduled execution. In GS,



(a) Response Time (b) Performance Profile

Figure 8: Impact of the number of CPUs (NN, WL5, p=16 (left bar) and 32 (right bar) , MPL=5, s=20%)

the idle component of the execution goes down with the system size, which may appear somewhat counter-intuitive. This is because there is an optimization being performed wherein tasks are replicated across time quanta whenever there are idle slots as long as it does not affect the coscheduled execution of other jobs [49]. A job could thus occur in more than one row of the Ousterhout matrix provided it does not require any migration across nodes. Since the overall response times drop, this optimization contributes to lowering the idleness component of the system.

Another exercise that would be useful to a system administrator/designer towards configuring a given system for maximum rewards is: Does it make sense to partition a given cluster into smaller fragments and specifically assign jobs to these fragments, or would it be better to treat it as one big machine? To investigate this issue, we run two sets of experiments: the first is with a single cluster of 32 nodes, and the second with two clusters of 16 nodes each, and exercise them with the same workload. Figure 9 compares the response times with these two executions (right bar for 32 node cluster and left bar for the partitioned system). We can observe that in nearly all cases, the savings in the wait time offset any stretching of the execution time, suggesting that it is not a good idea to partition the cluster. Similar results were observed for other system sizes/configurations as well.

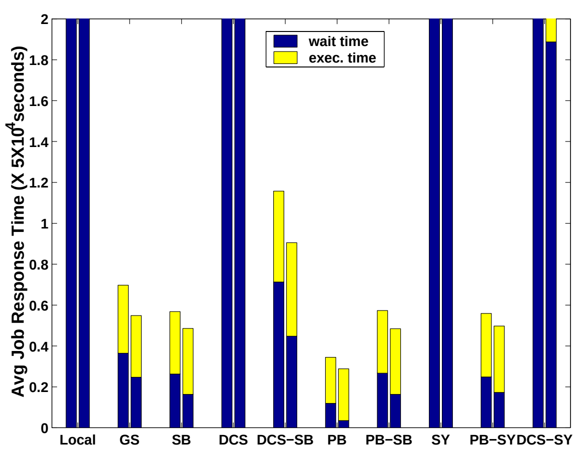
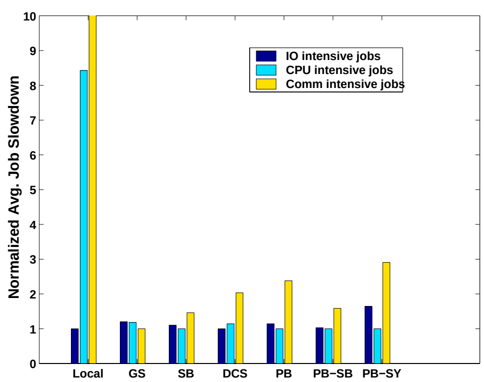


Figure 9: Impact of Partitioning on Response Time (NN, WL5, p=16 (left bar) and 32 (right bar) , MPL=5, s=20%)

4.8 Fairness

Until now, we have purely looked at performance issues for the different scheduling mechanisms. Fairness to different jobs is another important criteria from both the system administrator and user perspectives. There are several ways of analyzing the fairness of the system. In this paper, we look at the fairness issue from the viewpoint of how much the scheduler is biased/discriminatory based on the nature of the job. If a scheduler is biased towards/against any particular type of job, then it is likely to experience a lower/higher slowdown. So, comparing the differences in slowdown (or rather the coefficient of variation of slowdown) to the different job classes in a mixed workload can be used to argue the fairness of a scheduling strategy. To examine the fairness for different job types, we run workload WL8 (with s=0, to ensure that all job types have the same average execution time on a dedicated system) on a 32 node system, and classify the jobs within this workload into those that are CPU intensive, communication intensive, and I/O intensive. Figure 10 shows the average job slowdown (normalized for each with respect to the job class with the least slowdown) and its coefficient of variation over these three job classes for Local, GS, SB, DCS, PB, PB-SB, and PB-SY.

As expected, GS is the most fair, with all three job classes experiencing equal slowdowns (low coefficient of variation). At the other end of the spectrum is Local, where I/O intensive jobs get boosted much more often than the others (I/O completion is the only criterion for boost in priority in the Solaris scheduler at each node). Of the remaining schemes, the blocking schemes (SB and PB-SB) are more fair than their spinning counterparts (DCS, PB and PB-SY). We will revisit the fairness issue for the PB mechanism again in the next subsection after we discuss the different heuristics that one can use for PB.



(a) Slowdown

Scheme	Coeff. of Var.
Local	1.632
GS	0.099
SB	0.203
DCS	0.403
PB	0.504
PB-SB	0.274
PB-SY	0.524

(b) Coeff. of Variation of Slowdown

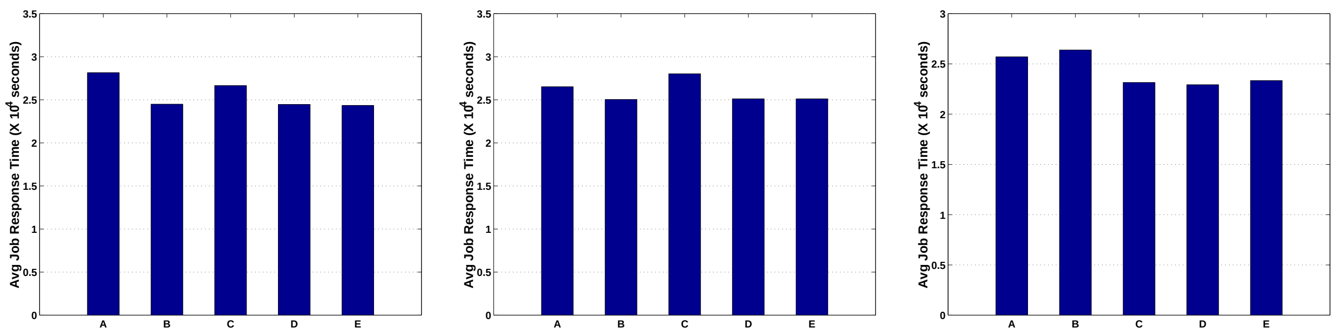
Figure 10: Fairness: Slowdown of Different Job Types (NN, WL8, p=32, MPL=5, s=0%)

4.9 PB Heuristics

The PB mechanism is a powerful one, allowing us to employ several heuristics to decide on who to boost before the next scheduling decision. Until now, we have used only one heuristic for this mechanism, and it will be interesting to explore alternatives for this choice (particularly if we can do even better). When the PB mechanism is invoked, a ready/executing process can be in one of 4 states: *S1*: It is in the compute/send phase and there are no pending (unconsumed) messages; *S2*: It is in the compute/send phase and there is at least one pending message; *S3*: It is in the receive phase and the message for which this receive is posted has arrived (and not yet consumed); and *S4*: It is in the receive phase and the message for which it is waiting has not yet arrived. Identifying these states can help us decide whether to boost a process, and to decide on a priority order if there are choices. Of these four states, it intuitively makes the most sense to give the highest priority to *S3* and *S2*, since a pending message implies that the receiver should soon be scheduled as per the DCS heuristic. Another observation is that a process in S4 should never get boosted since it will waste CPU cycles spinning redundantly (so we can confine ourselves to states S1, S2, and S3). Based on the relative priority of these states, we identify the following heuristics:

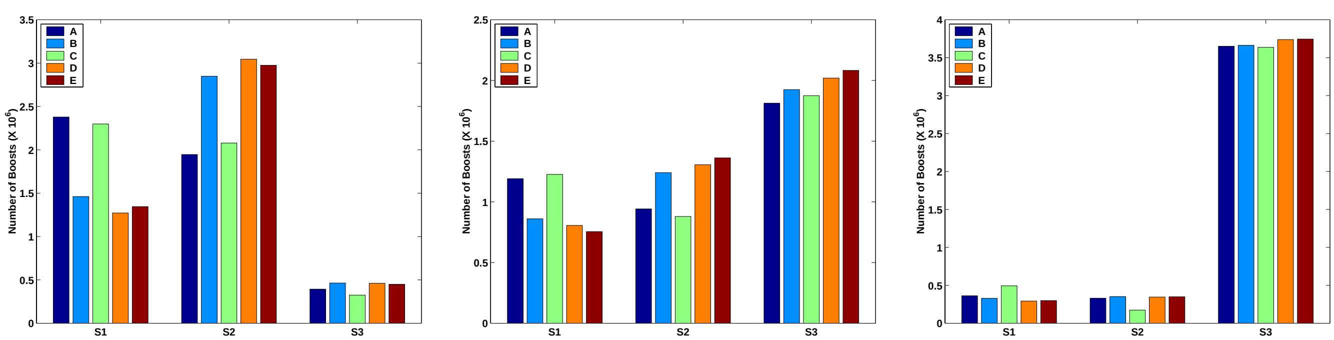
A: $S3 \rightarrow \{S2, S1\}$
B: $S3 \rightarrow S2 \rightarrow S1$
C: $\{S3, S2, S1\}$
D: $\{S3, S2\} \rightarrow S1$
E: $S2 \rightarrow S3 \rightarrow S1$

where $\rightarrow$ denotes the priority ordering. For instance, scheme **A** denotes that all processes are first checked to see if any of them is in state *S3* in round-robin order, stopping (and boosting) at the first process in this state. Only when none of them are in state *S3*, are the conditions *S2* and *S1* evaluated for all of them (again in round robin order) to see if a process is in either of these two states (and if so stopping and boosting that process). The other schemes can be similarly described. In all the previous performance results, heuristic **A** has been used.



(a) WL4 (b) WL8 (c) WL5

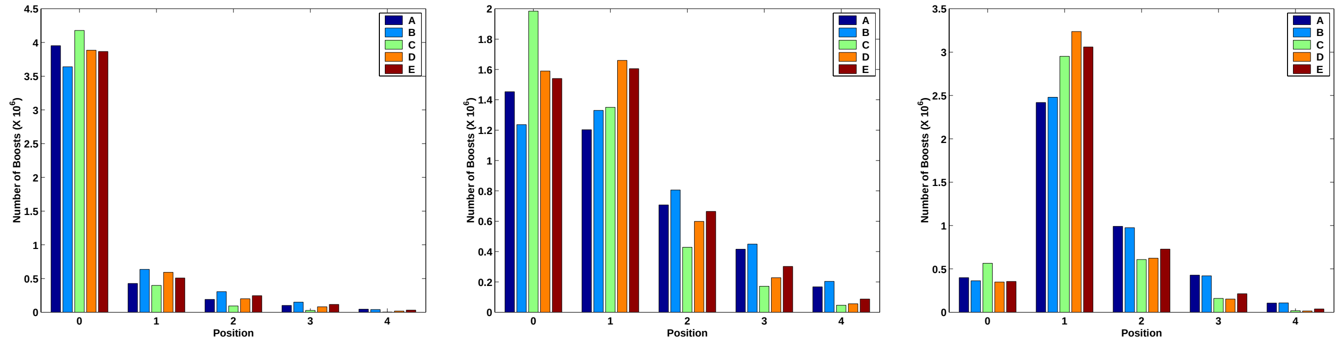
Figure 11: PB Heuristics (NN, p=32, MPL=5, s=20%)



(a) WL4 (b) WL8 (c) WL5

Figure 12: Conditions causing Boosts (NN, p=32, MPL=5, s=20%)

Figure 11 compares the response times of the five heuristics for the WL4 (CPU intensive), WL8 (mixed) and WL5 (communication intensive) workloads. For WL4, we find that heuristics **B**, **D** and **E** outperform the other two. Even though the workload is CPU intensive, it is still important to boost processes with one or more pending



(a) WL4 (b) WL8 (c) WL5

Figure 13: Position of Process being Boosted (NN, p=32, MPL=5, s=20%)

messages (to increase the likelihood of coscheduling). Of the five heuristics, **B**, **D** and **E** are likely to give more boosts to such processes than the others. The likelihood of **S3** being the chosen condition for boosting a process is rather low in this workload (being CPU intensive). In **A** and **C**, S1 has a higher chance (than in **B**, **D** and **E**) of being the condition used for boosting (because of the priority order). This intuition is also confirmed by the statistics from the experiments shown in Figure 12 (a) which gives the number of boosts given by the different heuristics based on the condition causing the boosts.

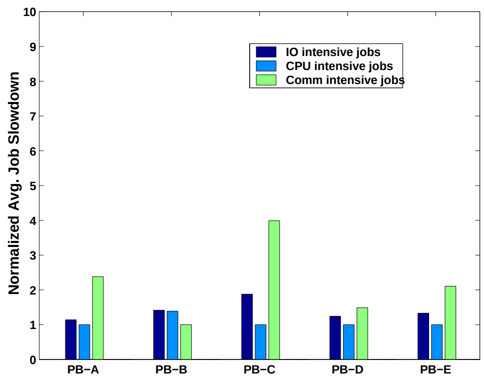
We find a similar argument (Figure 11 (b)) holding for the mixed workload (WL8) as well, where we find that heuristics **B**, **D** and **E** give more boosts to a job with a pending message (states S2 or S3) than the other two (Figure 12 (b)).

Moving to WL5, we find that heuristics **C**, **D** and **E** are better than the first two which can be explained as follows. For this workload, S3 is most likely to be the chosen condition for boosting a process, since it is very likely that the current process is in S4 state when the periodic boost mechanism is invoked, and the next immediate process in round-robin order is chosen (which is likely to be in S3 state). This intuition is borne out by the statistics shown in Figure 12 (c) where S3 gives the maximum number of boosts, and in Figure 13 (c) which shows the number of boosts given to a process that is i positions away (on the x-axis) from the currently scheduled process in round-robin order. In this workload, it is more important to adhere to some order that is common across the different CPUs to increase the likelihood of the processes belonging to the same job being scheduled at the same time (since they are all likely to be in the same state). Of the five heuristics, we find the latter three have a higher likelihood of selecting the next process in round-robin order, and thus have a lower response time.

In essence, two factors are to be considered in the performance of these heuristics. First is the condition causing the boost, and the second being the position (from the current process in round robin order) of the process that is boosted. In WL4 and WL8, the first factor is more important since the processes are likely to be in different states, while the latter factor is more important in WL5 where all (or most) the other processes at a node are likely to be in the same state (namely S3).

Overall, we find that heuristics **D** and **E** perform the best across all three workloads. We have also tried heuristics that are based on the number of pending messages and time-stamp of message arrivals, but we have still found **D** and **E** outperforming those as well.

We next examine the fairness of the different heuristics using the same experiment mentioned in Section 4.8. As before, Figure 14 shows the normalized job slowdown and its corresponding coefficient of variation respectively, for heuristics **A** through **E**. We find that **B** and **D** are much more fair than the others, with their coefficient of variation comparable to that of SB (Figure 10). We noticed that **D** was giving low response times as well, suggesting that we use this heuristic for the PB mechanism.



(a) Slowdown

Scheme	Coeff. of Var.
PB-A	0.504
PB-B	0.184
PB-C	0.671
PB-D	0.196
PB-E	0.384

(b) Coeff. of Variation of Slowdown

Figure 14: PB Fairness: Slowdown of Different Job Types (NN, WL8, p=32, MPL=5, s=0%)

The fairness issue suggests an even further improvement to the five PB heuristics, where instead of just checking the condition (S1 through S4) for boosting, we also ensure that the selected process does not get more than a fair share of the CPU (else an alternate is chosen). So we can revise the five heuristics with this additional check, and we refer to these new mechanisms as PB-A$'$ through PB-E$'$ respectively. For instance, PB-E$'$ can be explained as follows. First, we find all processes that are in S2 state. If there is at least one such process, then we use the process with the minimum ratio of the CPU time that it has been allocated to the time since it entered the system (i.e. percentage of the CPU that it has received since it started) as the target for boosting. However, if no process is in S2 state, then we check for S3 state and do likewise. Table 6 shows the coefficient of variation of these five variations, and it can be seen that the first four of these are as fair as GS (compare to Figure 10). Further, the response time of the enhanced mechanisms are plotted in Figure 15 in relation with the plain **A** through **E** heuristics, to show that we are not loosing out on response times at the cost of fairness.

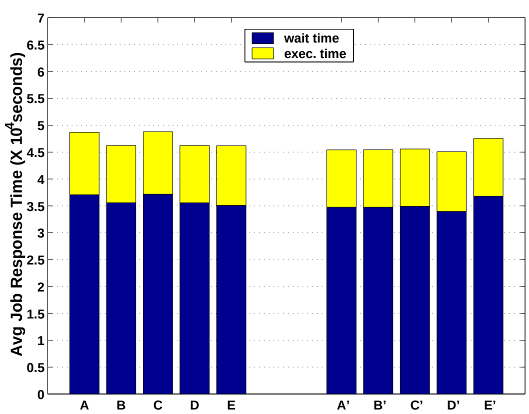


Figure 15: PB$'$: Response Time (NN, WL8, p=32, MPL=5, s=20%)

Scheme	Coeff. of Var.
PB-A$'$	0.101
PB-B$'$	0.123
PB-C$'$	0.082
PB-D$'$	0.058
PB-E$'$	0.280

Table 6: PB$'$: Coefficient of Variation of Slowdown (NN, WL8, p=32, MPL=5, s=0%)

5 Concluding Remarks and Future Work

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 [23, 8, 1, 37]. While our previous work [23] 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. This is evidenced by the poor performance of Local, which leaves it to the individual operating systems (on these nodes) to independently make their own decisions. Regardless of the operating condition (unless there is little or no communication at all), Local is not a viable option.

We have also observed that the dynamic coscheduling mechanisms are in general a better alternative to the conventional solution, Gang Scheduling (GS), employed today in many parallel machines [40, 3, 15, 39, 11]. In the first place, GS requires slots to be available on the required number of CPUs within the same time quantum (row of the Ousterhout matrix), which is not necessary in the dynamic coscheduling mechanisms. As a result, wait times in the arrival queue are typically higher in GS. 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); (c) the multiprogramming level is high (system overheads become more important); and (d) when the costs for explicitly synchronizing the nodes between time quanta becomes high (perhaps, for large systems). The one advantage that GS has over its dynamic coscheduling counterparts is that it is fair across different job types.

Comparing the Spin Block (SB), Spin Yield (SY) and Demand-based Coscheduling (DCS) approaches within the dynamic coscheduling domain, we find that blocking (SB) seems to be a better option most of the time. This is particularly true for: (a) higher multiprogramming levels (since the probability of finding alternate useful work on blocking is higher); (b) higher skewness (since they are less sensitive to skews between tasks); and (c) communication intensive workloads (since it better approaches coscheduling). The reason for the last observation was also made in [23], where it was mentioned that it is important to schedule the receiver of a message as early as possible while reducing spin times. DCS only addresses the first goal, and SY addresses only the second goal. SB, on the other hand, gives the blocked process a priority boost on message arrival, to attempt achieving both goals.

Uniformly, we find PB outperforming SB (and 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. While the performance of PB and SB are comparable for CPU and I/O intensive workloads, PB is much better when workloads are communication intensive. In addition, PB is preferable over SB for: (a) smaller MPLs (SB becomes comparable to PB only when the MPL is larger); and (b) higher system overheads (as switching/interrupt costs increase, the advantage of PB over SB is amplified since SB incurs more system overheads). It is only when the skewness gets large, does SB have a slight edge, and in such cases one could resort to a combination (PB-SB) of these mechanisms.

In this paper, we have also opened a new topic for further research that offers a lot of potential for improving PB even further. We have been able to identify a set of heuristics that can be used to decide on when and whom to boost, developed a classification of these heuristics, and have presented results showing the performance potential of this avenue of research. These heuristics take message arrival and process state into consideration in picking a candidate for a priority boost. Further, one could also augment these heuristics with a fair share (of the CPU) mechanism without compromising on the performance (response time) benefits. The resulting heuristic (PB-D′) has significantly lower response times than the other coscheduling alternatives, and is even more fair than GS.

While this paper is a comprehensive and novel exercise in the evaluation of numerous parallel coscheduling mechanisms using a unified framework, there are several interesting directions for future research. It would be interesting to theoretically model 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 [29] 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. 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. We have not considered memory constraints in this work, and incorporating such constraints to determine the multiprogramming level and associated overheads on the dynamic coscheduling strategies would also be very interesting [28, 16].

This study has taken an important step in improving the responsiveness of a high performance (parallel) computing platform, which have traditionally been optimized for throughput (for scientific applications). This can make it more suitable for the emerging use of clusters for web and database services. We propose to explore this issue using workloads from the emerging applications. Further, guaranteed (soft or hard) service [32, 36, 31, 17, 20, 5, 7] is also expected to become an important Quality-of-Service demand from such an environment, as clusters take on the challenges of multimedia and real-time applications. Extending the PB mechanism using well-known rate-based and real-time scheduling techniques is part of our ongoing and future work.

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. Banikazemi, V. Moorthy, D.K. Panda, L. Herger, and B. Abali. Efficient Virtual Interface Architecture Support for the IBM SP Switch-Connected NT Clusters . In *Proceedings of the International Symposium on Parallel and Distributed Processing Systems*, pages 33–42, May 2000.

[3] E. Barton, J. Cownie, and M. McLaren. Message Passing on the Meiko CS-2. *Parallel Computing*, 20(4), April 1994.

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

[5] B. Caminero, F. Quiles, J. Duato, D. Love, and S. Yalamanchili. Performance Evaluation of the MMR using MPEG-2 Video Traces . In *Proceedings of the Workshop on Communication, Architecture and Applications for Network Based Parallel Computing (CANPC)*, pages 62–76, January 1999.

[6] V. A. Dixit-Radiya and D. K. Panda. Clustering and Intra-Processor Scheduling for Explicitily-Parallel Programs on Distributed-Memory Systems . In *Proceedings of the International Parallel Processing Symposium*, pages 609–615, 1994.

[7] J. Duato, S. Yalamanchili, B. Caminero, D. Love, and F. Quiles. MMR: A MultiMedia Router: Architecture and Design Trade-offs . In *Proceedings of the IEEE Symposium on High Performance Computer Architecture*, January 1999.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] X. He and Q. Yang. Performance evaluation of distributed webservers under commercial workload. In *Proceedings of the Internet Conference*, 2000.

[13] 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.

[14] Y. Hu, A. Nanda, and Q. Yang. Measurement, Analysis and Performance Improvement of the Apache Web Server . In *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference (IPCCC'99), Phoenix/Scottsdale, Arizona*, February 1999.

[15] Intel Corporation, Oregon. *Paragon User's Guide*, 1993.

[16] N. Islam, A. L. Prodromidis, M. S. Squillante, L. L. Fong, and A. S. Gopal. Extensible Resource Management for Cluster Computing. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 561–568, 1997.

[17] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham. Integrated Scheduling of Multimedia and Hard Real-Time Tasks. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, June 1996.

[18] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 226–236, 1990.

[19] 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.

[20] D. Love, S. Yalamanchili, J. Duato, M.B. Caminero, M.B. Caminero F.J. Quiles, and F.J. Quiles. Switch Scheduling in the Multimedia Router (MMR) . In *Proceedings of the International Symposium on Parallel and Distributed Processing Systems*, pages 5–12, May 2000.

[21] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[22] 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.

[23] 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.

[24] 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.

[25] V. K. Naik, S. K. Setia, and M. S. Squillante. Performance analysis of job scheduling policies in parallel supercomputing enviroments. In *Proceedings of Supercomputing'93*, pages 824–833, November 1993.

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

[27] 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.

[28] V. G. J. Peris, M. S. Squillante, and V. K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 5–18, May 1994.

[29] F. Petrini and W. Feng. Buffered Coscheduling: A New Method for Multitasking Parallel Jobs on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 439–444, May 2000.

[30] R. Poovendran, P. Keleher, and J. S. Baras. A Decision-process Analysis of Implicit Coscheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1115–120, May 2000.

[31] K. Ramamritham, C. Shen, O. Gonzalez, S. Sen, and S. B Shirgurkar. Using Windows NT for Real-Time Applications:Experimental Observations and Recommendations. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications*, June 1998.

[32] K. Ramamritham and J. A. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. *Proceedings of the IEEE*, pages 55–67, January 1994.

[33] S. K. Setia, M. S. Squillante, and S. K. Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel computers. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 158–170, May 1993.

[34] S. K. Setia, M. S. Squillante, and S. K. Tripathi. Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):401–420, April 1994.

[35] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2-3):107–140, March 1994.

[36] C. Shen, O. Gonzalez, K. Ramamritham, and I. Mizunuma. User Level Scheduling of Communication for Real-Time Tasks. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications*, June 1999.

[37] 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.

[38] 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.

[39] K. Suzaki and D. Walsh. Implementing the Combination of Time Sharing and Space Sharing on AP/Linux. In *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pages 83–97, March 1998. LNCS 1459.

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

[41] A. Tucker. *Efficient Scheduling on Shared-Memory Multiprocessors*. PhD thesis, Stanford University, November 1993.

[42] L. W. Tucker and G. G. Robertson. Architecture and Applications of the Connection Machine. *IEEE Computer*, 21(8):26–38, August 1988.

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

[44] 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.

[45] F. Wang, M. Papaefthymiou, and M. Squillante. Performance Evaluation of Gang Scheduling for Parallel and Distributed Multiprogramming. In *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pages 277–298, April 1997. LNCS 1291.

[46] L. Xiao, X. Zhang, and Y. Qu. Effective load sharing on heterogeneous networks of workstations . In *Proceedings of the International Symposium on Parallel and Distributed Processing Systems*, May 2000.

[47] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 214–225, 1990.

[48] X. Zhang, Y. Qu, and L. Xiao. Improving distributed workload performance by sharing both CPU and memory resources . In *Proceedings of the International Conference on Distributed Computing System (ICDCS'00)*, April 2000.

[49] 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*, pages 133–142, May 2000.

[50] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke. A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment. In *Proceedings of the ACM 2000 International Conference on Supercomputing*, pages 100–109, May 2000.