

# Coscheduling in Clusters: Is It a Viable Alternative? \*

Gyu Sang Choi<sup>1</sup>   Jin-Ha Kim<sup>1</sup>   Deniz Ersoz<sup>1</sup>   Andy B. Yoo<sup>2</sup>   Chita R. Das<sup>1</sup>

<sup>1</sup>Dept. of Computer Science and Engineering   <sup>2</sup>Lawrence Livermore National Laboratory  
Penn State University   Livermore, CA 94551  
University Park, PA 16802  
e-mail: {gchoi, jikim, ersoz, das}@cse.psu.edu   e-mail: {yoo2}@llnl.gov

## ABSTRACT

In this paper, we conduct an in-depth evaluation of a broad spectrum of scheduling alternatives for clusters. These include the widely used batch scheduling, local scheduling, gang scheduling, all prior communication-driven coscheduling algorithms (*Dynamic Coscheduling* (DCS), *Spin Block* (SB), *Periodic Boost* (PB), and *Co-ordinated Coscheduling* (CC)) and a newly proposed *HYBRID* coscheduling algorithm on a 16-node, Myrinet-connected Linux cluster.

Performance and energy measurements using several NAS, LLNL and ANL benchmarks on the Linux cluster provide several interesting conclusions. First, although batch scheduling is currently used in most clusters, all blocking-based coscheduling techniques such as SB, CC and HYBRID and the gang scheduling can provide much better performance even in a dedicated cluster platform. Second, in contrast to some of the prior studies, we observe that blocking-based schemes like SB and HYBRID can provide better performance than spin-based techniques like PB on a Linux platform. Third, the proposed HYBRID scheduling provides the best performance-energy behavior and can be implemented on any cluster with little effort. All these results suggest that blocking-based coscheduling techniques are viable candidates to be used in clusters for significant performance-energy benefits.

## Categories and Subject Descriptors

D.4.1 [Process Management]: Scheduling; D.4.8 [Performance]: Measurements; C.2.4 [Distributed Systems]: Distributed Applications

## General Terms

Experimentation, Performance Measurement, Scheduling

\*This research was supported in part by NSF grants CCR-9900701, CCR-0098149, EIA-0202007 and CCR-0208734.  
0-7695-2153-3/04 \$20.00 (c)2004 IEEE

## Keywords

Coscheduling, Linux Cluster, Myrinet, Batch Scheduling, Gang Scheduling, Energy Consumption

## 1. INTRODUCTION

Cluster systems [4], built using commodity off-the-shelf (CO-TS) hardware and software components, are becoming increasingly more attractive for supporting a variety of scientific and business applications. Most supercomputer platforms, university computing infrastructures, data centers and a myriad of other applications use clusters [45]. These clusters are used either as dedicated Beowulf systems or as non-dedicated time-sharing systems in solving parallel applications. As the wide-spread use of these systems spanning from a few nodes to thousands of nodes (like IBM ASCI [28] and MCR Linux Cluster [45] in LLNL) continues, improving their performance and energy efficiency becomes a critical issue. While improving the performance has always been the main focus, some recent studies indicate that maintenance of these data centers and supercomputers requires enormous amount of power [27]. Typical power requirements range from about 1.5 megawatt for a data center to about 18 megawatts for some of the supercomputers [7, 25, 45]. A more recent projection for running a PFlop system is about 100 megawatts [7]. A conservative estimate of the annual operational budget of a PFlop system (assuming \$10 per megawatt) comes to about millions of dollars! Thus, even a small reduction in the energy consumption without compromising the deliverable performance will have significant financial impacts.

One viable approach to enhance the performance-energy behavior of clusters is to use an efficient scheduling algorithm, since it has a direct impact on system performance. The possible alternatives span from simple batch scheduling [15] and native local scheduling to more sophisticated techniques like gang scheduling [15, 16, 18, 14] or communication-driven coscheduling [41, 6, 32]. Most of the universities, government labs and commercial installations still use some form of batch scheduling [19, 1] for their research, development, and testing clusters.

In local scheduling, processes of a parallel job are independently scheduled by the native OS scheduler on each node, without any effort to coschedule them. Although simpler to implement, local scheduling can be very inefficient for running parallel jobs that need process coordination. Gang scheduling, on the other hand, uses explicit global

synchronization to schedule all the processes of a job simultaneously, and has been successfully deployed in commercial Symmetric Multi-Processing (SMP) systems. To some extent, this has been shown viable for dedicated clusters [18, 14]. Recently, a few coscheduling alternatives such as Dynamic Coscheduling (DCS) [41], Implicit Coscheduling (ICS) [6], Spin Block (SB) [32], Periodic Boost (PB) [32] and Co-ordinated Coscheduling (CC) [3] have been proposed for clusters. These coscheduling techniques rely on the communication behavior of the applications to schedule the communicating processes of a job simultaneously. Using efficient user-level communication protocols such as U-Net [13], Active Messages [46], Fast Messages [34] and VIA [12], coscheduling has been shown to be quite efficient in a time-sharing environment. However, to the best of our knowledge, none of these coscheduling techniques have yet made their way into real deployments.

The motivation of this paper is to enhance the deliverable performance and reduce the energy consumption of dedicated clusters through efficient scheduling. In this context, we attempt to investigate the following issues: (i) *How do the communication-driven coscheduling techniques compare against the batch scheduling?*; (ii) *How do the coscheduling techniques compare against gang scheduling?*; (iii) *Can we design/identify some coscheduling techniques, which can be used instead of a batch scheduler to enhance performance-energy behavior?*; and (iv) *What about the implementation, portability and scalability issue for deploying these schemes on different hardware and software platforms?*

To address these concerns, we conduct an in-depth evaluation of a broad spectrum of scheduling alternatives for clusters. These include a widely used batch scheduler (PBS) [1], local scheduling, gang scheduling, all prior communication-driven coscheduling algorithms, and a newly proposed *HYBRID* coscheduling algorithm. In order to provide ease of implementation and portability across many cluster platforms, we propose a generic framework for deploying any coscheduling algorithm by providing a reusable and dynamically loadable kernel module. We have implemented four prior coscheduling algorithms (*Dynamic Coscheduling* (DCS), *Spin Block* (SB), *Periodic Boost* (PB), and *Co-ordinated Coscheduling* (CC)) and the *HYBRID* coscheduling using this framework on a 16-node, Myrinet-connected [9] Linux cluster that uses GM as the communication layer. We have ported a previously proposed gang scheduler (SCore) [18] to our platform for comparison with other schemes.

Unlike the other coscheduling schemes, our proposed *HYBRID* coscheduling scheme adopts a mixed approach of the gang scheduling and communication-driven coscheduling. Like gang scheduling, it attempts to coschedule the parallel processes of a job. However, the coscheduling is not explicit as in the gang scheduling and the processes need not be coscheduled for the entire execution time. Similarly, like communication-driven coscheduling schemes, the *HYBRID* scheme tries to approximately coschedule only the communicating processes. But, unlike the former techniques, it does not depend on the explicit message arrival or waiting hint for coscheduling. We achieved this by applying different scheduling policies to the computation and communication phases of a program. We boost the priority of all the

communicating processes at the beginning of a communication phase hoping that they are all coscheduled, and lower their priority at the end of communication. This is done at the MPI level by detecting the communication activities. For the computation phase, the processes are scheduled by the local Linux scheduler. The main advantage of the *HYBRID* coscheduling scheme is its simplicity and portability. Since this scheme is implemented at the parallel program level (e.g.: MPI), it does not require modifications to a NIC firmware and user-level communication. Thus, it can be ported to another platform with little effort.

We conduct an extensive measurement-based study of all scheduling techniques using several NAS, LLNL and ANL benchmarks on the Linux cluster. The main objective functions here are the average completion time and total energy consumption. For the power/energy measurements, we use the WT210 [47] measurement tool. In addition, we also analyze the impact of memory swapping on coscheduling algorithms. We believe that this is the first effort that not only does a comprehensive performance analysis of all scheduling techniques, but also considers power and other factors into the overall equation.

This paper provides several interesting conclusions. First and foremost, the most important observation is that we can achieve significant performance improvements by deploying a suitable communication-driven coscheduling instead of the widely used batch scheduling. For example, schemes like *HYBRID*, *CC* and *SB* can reduce the average completion time orders of magnitude compared to a batch scheduler under heavy load. Second, blocking-based schemes like *HYBRID*, *CC* and *SB* perform significantly better than the spin-only based schemes like *PB* or *DCS*, contrary to some previous results [32, 42]. They can provide competitive or even better performance than gang scheduling. Third, the proposed *HYBRID* scheme is the best performer and can be implemented on any communication layer with minimal effort. Fourth, it is possible to devise a memory-aware coscheduling technique, which can avoid expensive memory swapping, and can still perform better than a batch scheduler. Finally, the improved efficiency of a scheduling algorithm translates to energy conservation or better performance-energy ratio. Our experiments show that under heavy load, the *HYBRID* scheme is 38% better energy-efficient compared to the batch scheduler. All these performance and energy results strongly support the case for using a coscheduling algorithm in dedicated clusters.

The rest of this paper is organized as follows: In Section 2, we provide a summary of all the prior scheduling techniques for clusters. Section 3 outlines the generic framework that can be used for implementing any coscheduling scheme. The *HYBRID* coscheduling algorithms is explained in Section 4. The performance results are analyzed in Section 5, followed by the concluding remarks in the last Section.

## 2. BACKGROUND AND RELATED WORK

In this section, we summarize batch scheduling, gang scheduling and communication-driven coscheduling techniques.

### 2.1 Batch Scheduling

Batch scheduling is the most popular policy to manage dedicated clusters for running non-interactive jobs. NQS [15], Maui [43], IBM LoadLeveler [19], and PBS [1] are widely-used batch schedulers. OpenPBS and PBSPro are the open source version and commercial version of PBS, respectively. Typically, a batch scheduler is used for large scientific applications to avoid memory swapping. The disadvantages of batch scheduling are low utilization and high completion time [39]. To solve these problems, various backfilling techniques [49, 29] have been proposed, where a job, which is not at the head of the waiting queue, is allowed to run by bypassing other jobs. In this paper, we use the OpenPBS in our cluster system.

## 2.2 Gang Scheduling

The gang scheduling (Explicit coscheduling) is an efficient coscheduling algorithm for fine-grained parallel processes. It has two features: (i) all the processes of a parallel job, called a gang, are scheduled together for simultaneous execution using one-to-one mapping, and (ii) context switching is coordinated across the nodes such that all the processes are scheduled and de-scheduled at the same time. The advantage of a gang scheduler is faster completion time because the processes of a job are scheduled together, while its disadvantage is the global synchronization overhead needed to coordinate a set of processes. Gang scheduling has been mainly used in supercomputers (GangLL [23] is currently used in ASCI machines.). SCORE-D [18] is the first implementation of a gang scheduler in a Linux cluster.

## 2.3 Communication-Driven Coscheduling

Unlike the gang scheduling, with a communication-driven coscheduling like DCS [41], SB [32], PB [32] or CC [3], each node in a cluster has an independent scheduler, which coordinates the communicating processes of a parallel job. All these coscheduling algorithms rely primarily on one of the two local events (*arrival of a message* and *waiting for a message*) to determine when and which process to schedule. For example, in SB [6, 32], a process waiting for a message spins for a fixed amount of time before blocking itself, hoping that the corresponding process is coscheduled at the remote node. DCS [41] uses an incoming message to schedule the process for which the messages are destined. The underlying idea is that there is a high probability that the corresponding sender is scheduled at the remote node and thus, both processes can be scheduled simultaneously. In the PB scheme [32], a periodic mechanism checks the endpoints of the parallel processes in a round-robin fashion and boosts the priority of one of the processes with un-consumed messages based on some selection criteria. The recently proposed CC scheme [3] is different in that it optimizes the spinning time to improve performance at both the sender and receiver. With this scheme, the sender spins for a pre-determined amount of time before blocking, waiting for an acknowledge from the Network Interface Controller (NIC). On the receiver side, a process waits for a message arrival within the spin time. If a message does not arrive within this time, the process is blocked and registered for an interrupt from the NIC. In a regular interval, a process that has the largest number of un-consumed incoming message, is scheduled to run next.

All prior coscheduling schemes have been evaluated on non-dedicated small clusters and have arrived at different conclu-

sions. For example, PB was shown to be the best performer on a Solaris cluster [32], while CC and SB provided better performance than PB and DCS on a Linux cluster [3]. However, none of these techniques have been evaluated against the batch and gang schedulings to determine their feasibility for use in dedicated clusters.

## 3. A COSCHEDULING FRAMEWORK

To implement a communication-driven coscheduling scheme, we usually need to modify the NIC’s device driver and firmware, and the user-level communication layer. As shown in Figure 1 (a), a scheduling *policy* is mainly implemented in the device driver to decide which process should be executed next (typically by boosting its priority). Next, a *mechanism* module is implemented in the NIC firmware and user-level communication layer to collect the information required by the device driver. Thus, implementation of a coscheduling algorithm requires significant amount of effort and time. Moreover, this effort needs to be repeated whenever we move to a new platform.

This observation motivates us to propose a generic framework in which, a *policy* can be standardized and reused as a stand-alone kernel module, and proper interfaces can be outlined to implement the boosting *mechanisms* in the firmware, whenever required. Thus, the overall idea is to cleanly abstract a coscheduling *policy* from its underlying implementation *mechanism* so that both can be treated independently.

We now describe implementation of our scalable and reusable framework in a bottom-up fashion. As shown in Figure 1 (b), our design logically comprises of two layers. At the lowest layer (layer 1), we implemented a kernel scheduler patch (Linux 2.2, Linux 2.4) that provides flexibility for the system software developers to change their local scheduling policies through an independently loadable kernel module, which is built upon a previous effort by Rhine at HP labs [37]. Next, we developed a dynamically loadable kernel module [36], called *SchedAsst*, at layer 2, and several re-usable coscheduling *policies* are implemented in this module. Every time the local Linux scheduler is invoked by the system, before making a selection from its own *runqueue* [10], the *SchedAsst* selects the next process to run based on certain *criteria*, and returns its recommendation to the native scheduler. The *criteria* for selection is our *policy*, and is clearly specific to the coscheduling technique enforced. The native scheduler *optionally* verifies the recommendation of the *SchedAsst* for fairness before making a final selection decision. The final decision is then conveyed back (optionally) to the *SchedAsst* for its book-keeping.

This design clearly gives us several advantages. First, our framework can be reused to deploy and test several coscheduling mechanisms on a Linux cluster with minimal effort. Second, the coscheduling module can be tied in easily with any user-level communication infrastructure (VIA, IBA, U-Net, GM, FM, AM etc.), providing us the flexibility and generality. Finally, by adhering to the standard interface for implementing a coscheduling *mechanism*, the driver/firmware writers can easily and independently provide coscheduling support.

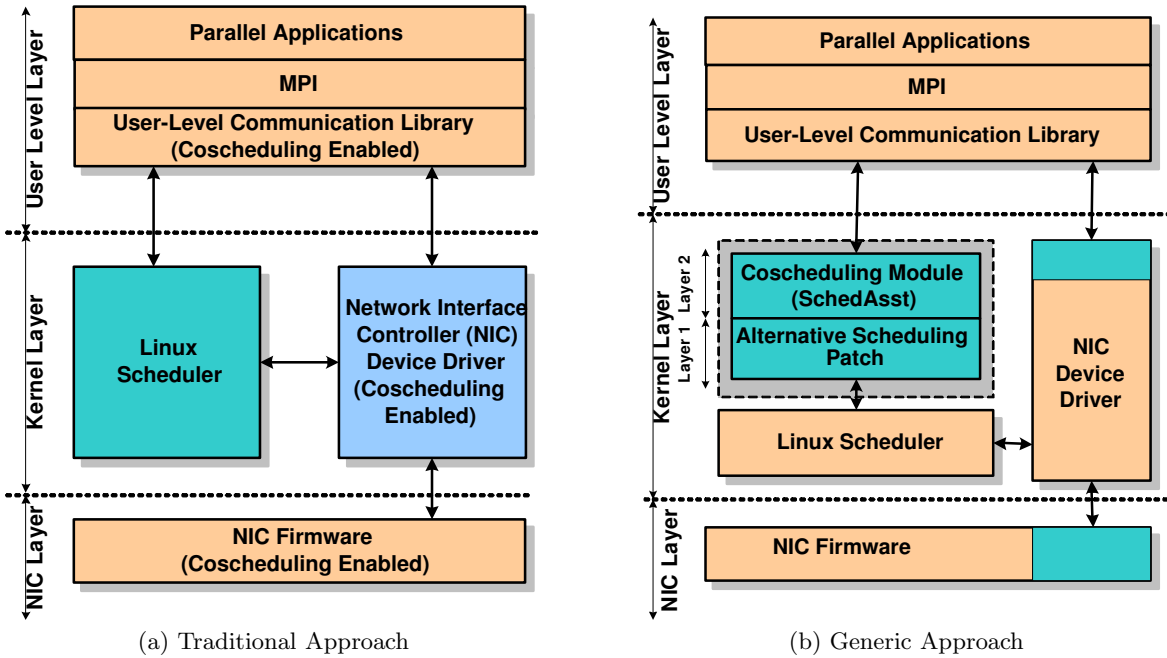


Figure 1: Traditional and the Generic Coscheduling Frameworks

Using the framework, we have implemented four prior coscheduling schemes (DCS, PB, SB and CC) and the newly proposed HYBRID coscheduling, which will be discussed in the next Section. The coscheduling *policies* are implemented in *SchedAsst* in Figure 1 (b). We also modified the GM’s [31] device driver and firmware to add the *mechanism* modules to gather communication information. GM has one completion queue for all acknowledgments from a NIC; including the *send* completion and *recv* completion. To implement SB correctly, we had to divide the single completion queue of GM into a send completion queue and a receive completion queue, and use the SB mechanism only for the *recv* completion queue. For DCS, we added a module in the NIC firmware that compares a current process to the corresponding process of the message and notifies the discrepancy to the *SchedAsst*. All these implementations in GM required considerable effort.

#### 4. HYBRID COSCHEDULING

We propose a new scheme, called HYBRID coscheduling, which combines the intrinsic merits of both gang scheduling and communication-driven coscheduling. Execution of a parallel process consists of two phases; computation and communication. A gang scheduler coschedules all the processes of a parallel job during its entire execution, and thus, suffers from high global synchronization overhead. Ideally, global synchronization is not necessary during its computation phase. On the other hand, all communication-driven coscheduling schemes try to approximately coschedule the processes only in the communication phase. For example, DCS and PB use a boosting mechanism for the process, which receives a message from the corresponding node and SB employs a spin-block mechanism at the receiver side. Thus, unlike the gang scheduler, coscheduling is implicit here.

The HYBRID coscheduling imitates the gang scheduling policy without using any global synchronization when a process is in the communication phase. We achieve this by explicitly boosting the priority of a process locally when it enters a collective communication phase, as illustrated in Figure 2, hoping that all other corresponding processes are also coscheduled. The processes return to the normal state at the end of the communication phase. We implemented this boosting mechanism in the *SchedAsst* of our coscheduling framework.

The proposed HYBRID scheme consists of two techniques. First, an immediate blocking is used at a sender and a receiver. It means that when a parallel process sends or receives a message, the process is immediately blocked if it is not complete. In prior blocking-based coschedulings, a parallel process spins for a fixed time (e.g. 300us) before being blocked. Second, when a parallel process enters a collective communication phase (e.g. MPI\_Bcast, MPI\_Barrier, etc), its priority is boosted until it leaves the communication phase. At the end of communication, its priority is lowered to its original value. All prior communication-driven coscheduling algorithms approximately coschedule each communication. In our proposed scheme, we try to coschedule all the communicating processes from the beginning to the end, not only an individual communication. In Table 1, we summarize the actions at the sender and receiver for all schemes when they wait for the completion of each communication, and the boosting policy at the beginning and end of a collective communication phase.

The HYBRID coscheduling algorithm introduces two design issues. One is to differentiate between the computation and communication phases in a parallel program and the other is the fairness problem caused by the boosting mechanism because priority boosting of a specific process is likely to cause

Coscheduling Scheme	Receiver	Sender	Collective Communication Phase	
			Beginning	End
DCS	Spin-only, Boost a priority	Spin-only	Nothing	Nothing
SB	Spin-Block	Spin-only	Nothing	Nothing
PB	Spin-only, Boosting	Spin-only	Nothing	Nothing
CC	Spin-Block, Boost a Priority	Spin-Block, Boost a Priority	Nothing	Nothing
HYBRID	Immediate Block	Immediate Block	Boost a priority	Restore a priority

Table 1: Actions taken by the Communication-driven Coscheduling Algorithms

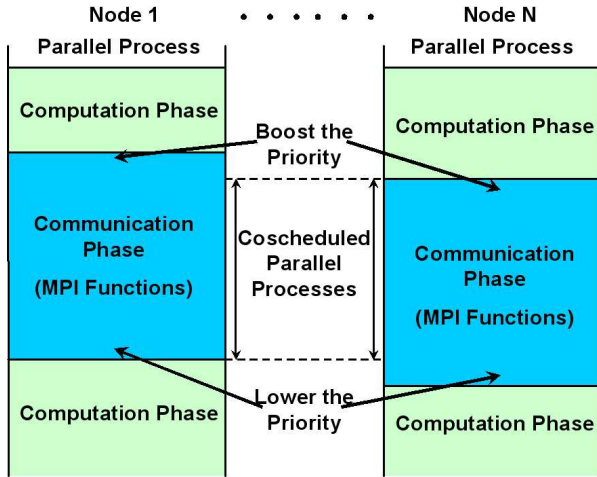


Figure 2: HYBRID Coscheduling Mechanism

starvation of the other processes in the same node. First, to schedule a process differently depending on the computation and communication phases, we add the boosting code only in the collective communication functions of the Message Passing Interface (MPI) (e.g.: `MPIBarrier`, `MPIAlltoall`, `MPIAllreduce`, etc.), which is used as the message passing layer in our programming model. It is also possible that a compiler recognizes the communication part in a parallel program and inserts the boosting code automatically at compile time to get better performance. This is a future research topic. We do not insert the boosting code in point-to-point MPI functions because these are faster than collective communication and thus, the performance gain will be minimal. Second, to guarantee fairness, the HYBRID coscheduling uses the immediate blocking mechanism. Therefore, a process that waits for a reply, is immediately blocked if the message has not been received.

The first advantage of the HYBRID coscheduling scheme is its simplicity and portability. The HYBRID scheme can be implemented simply at the parallel program level. Thus, it can be deployed on a communication layer such as VIA, GM and TCP/IP without requiring any modifications of the system software. Second, the proposed policy eliminates many overheads of other communication-driven coscheduling schemes. A typical communication-driven coscheduling scheme monitors the communication activities and a coscheduling module decides on which process to be boosted based on the communication activity. Thus, it incurs some processing overhead, while the HYBRID coscheduling can

eliminate this overhead. In DCS, when a NIC receives a message, it checks the current running process. If the message is not destined for the current running process, the NIC raises an interrupt for an appropriate process. The number of interrupts would be high for fine-grained communication. Third, the HYBRID mechanism eliminates the possible delay between a message arrival at a node and the actual boosting of the process, because the priority is already boosted whenever a process enters the communication phase. In SB and CC, a receiver process waits for a pre-determined period for a reply hoping that the corresponding process at a remote node is also currently running. If the reply does not arrive within this time, it releases the CPU and blocks itself. Hence, the CPU time can be wasted spinning if the sender and receiver processes are not coscheduled. This penalty can be eliminated in HYBRID coscheduling because it blocks the process immediately if there is no reply from the corresponding node.

## 5. PERFORMANCE EVALUATION

In this section, we evaluate the performances of four types of scheduling techniques (local scheduling, gang, batch and communication-driven coscheduling) on a 16-node cluster using NAS, LLNL and ANL benchmarks. In addition, we investigate the memory swapping effect and measure the energy consumption of each scheduling scheme.

### 5.1 Experimental Platform and Workload

Our experimental testbed is a 16-node Linux (version 2.4.7-10) cluster, connected through a 16-port Myrinet [9] switch. Each node is an Athlon 1.76 GHz uni-processor machine, with 1 GBytes memory and a PCI based on-board intelligent NIC [9], having 8 MB of on-chip RAM and a 133 MHz Lanai 9.2 RISC processor. We used Myrinet's GM implementation (version 1.6.3) [31] over Myrinet's NIC as our user-level communication layer and Myrinet's MPICH (version 1.2.5.9) implementation [30] over GM as our parallel programming library. GM supports two communication mechanisms; (i) non-blocking and (ii) blocking. In CC/SB and HYBRID schemes, we use the blocking mechanism of GM. On this platform, we measured application-to-application one-way latency to be around  $9.7\mu sec$ , averaged over 100,000 ping-pong short messages. This includes protocol processing overheads on both the sender as well as the receiver ends.

We use primarily NAS Parallel benchmarks (version 2.3) [33] to evaluate the performance of all scheduling schemes in this paper. The NAS Parallel benchmarks consist of eight applications and we use all programs except FT because it needs the F90 Fortran compiler that is not currently available in

our platform. In addition, we use the sPPM program from LLNL and BlockSolve95 code in ANL. BlockSolve95 [24] is a parallel program to solve sparse linear problems for physical models and sPPM [26] benchmark is a program to solve 3D gas dynamics models on a uniform Cartesian mesh.

For gang scheduling, we use Score-D (version 5.4) with PM (version 2) [44] as the user-level communication. On the same Myrinet NIC, we use GM for the batch scheduling (PBS) and communication-driven coschedulings and use PM for the gang scheduling (SCore). The performance of GM and PM is very close and thus, the performance comparison between gang scheduling and others is valid. (We compared the execution time of a single NAS application based on GM and PM, and the difference is less than 2%.)

NAS Benchmarks	Communication Intensity	Execution Time (sec)		
		Class A	Class B	Class C
CG	High	1.3	49.2	186.0
IS	High	1.0	2.4	10.9
SP	Medium	34.5	152.5	607.1
MG	Medium	2.4	61.5	191.0
BT	Low	53.1	435.3	995.4
LU	Low	32.2	148.7	604.9
EP	Very low	7.9	31.3	126.2

(a) NAS Benchmark Application

Benchmarks	Communication Intensity	Execution Time (sec)
sPPM	Low	2726
BlockSolve95	Medium	941

(b) sPPM and BlockSolve95

**Table 2: Execution Times of NAS Benchmarks, sPPM and BlockSolve95 (16-node Cluster)**

In order to better understand the results of this paper, we summarize the execution time of each NAS application from CLASS A to CLASS C, sPPM and BlockSolve95 on our 16-node cluster in Table 2. Moreover, Table 2 shows the communication intensity of each benchmark since the performance is related to the communication intensity in communication-driven coscheduling techniques.

In a real system, a parallel program arrives at a cluster and waits in the waiting queue if it cannot be allocated immediately. After a certain amount of queuing time, it is assigned to the required number of processors. We imitate the arrival pattern of jobs in a real system by randomly generating several jobs with varying problem sizes (CLASS A, B and C) and different number of processors per job (The number of processors is selected from 4, 8, 9 or 16 as per the NAS application requirements.). These jobs arrive at the cluster with exponentially distributed inter-arrival times (Inter-arrival time distribution analysis of LLNL and several university workloads [48] showed an exponential fit.). In this experiment, we consider the average completion time per job as the performance metric, where the completion time is the sum of the queuing time and execution time.

We consider two different types of job allocation: PACKING

and NO PACKING. In the NO PACKING case, parallel processes of a job are randomly allocated to the available nodes in the system, while in the PACKING scheme, contiguous nodes are assigned to a job to reduce the system fragmentation and increase the system utilization. Note that PACKING is the default configuration in PBS [1] and SCore [1, 18]. The Backfilling technique [49, 29], which is widely used in batch scheduling, is another method to reduce the system fragmentation. We performed experiments with the backfilling technique in our testbed and the results of PBS with and without backfilling are very close (within 2%). The reason backfilling did not have significant performance difference is because of the characteristics of the NAS benchmark. The backfilling method is very useful when the system has many fragments that can be used for small jobs. But the number of required processors for the NAS benchmarks is usually  $2^N$  or  $N^2$ . In addition, our 16-node Linux cluster is relatively small to capture the backfilling effect.

In our experiments, we limit the maximum Multi Programming Level (MPL) to three for the gang scheduling and all communication-driven coschedulings, and we limit the total size of simultaneously running programs to fit the memory (1GBytes) because our default analysis does not consider memory swapping. The memory swapping issue is dealt with in the next section separately. We also implemented an allocator for the communication-driven coscheduling algorithms, and for the batch and gang schedulings, we use the allocation modules provided by PBS and SCore.

## 5.2 Comparisons of All Scheduling Techniques

Figure 3 depicts the average completion time comparison of all scheduling techniques under heavy and moderately light workloads. The results are obtained by running 100 mixed applications from the NAS benchmarks as described earlier. In addition, we also collect results with and without packing. Figure 3 (a) depicts the completion time results when the system is highly loaded (the average job inter-arrival time is 100 seconds), whereas the results of Figure 3 (b) are obtained in a lightly loaded condition (the average job inter-arrival time is 250 seconds). Performance measurements on ASCI Blue-Pacific at LLNL [48] with gang scheduling indicated that the system under heavy load experiences long queuing time, which could be as high as six times of the execution time. Thus, our heavy load experiments represent a typical dedicated cluster environment in a smaller scale.

In Figures 3 (a) and (b), we see that the PACKING scheme helps improving the performance as expected. The average performance gain with PACKING in Figure 3 (a) is about 20% compared to the NO PACKING case. In the batch scheduling (PBS) [1] and gang scheduling (SCore) [18], we do not have the results without PACKING, since these two schemes use PACKING as the default mode. In Figure 3 (a), the result of the Native Local Scheduler (NLS) with NO PACKING shows the worst completion time and PB with NO PACKING and PBS follow next. The batch scheduling (PBS) has the longest waiting time and this confirms the weakness of a batch scheduler.

As expected, in Figure 3 (a), the completion time of each scheduling scheme is much longer than that in Figure 3 (b). In Figure 3 (a), the completion times increase due to large

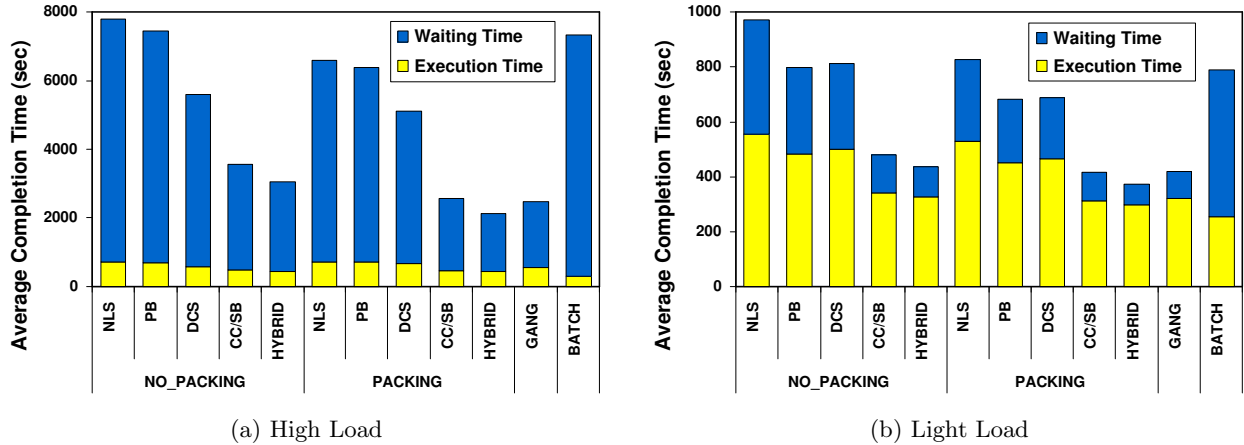


Figure 3: Performance Comparison of Different Scheduling Techniques (100 jobs, No Memory Swapping, MPL=3)

waiting times, but in Figure 3 (b), the difference between the execution times of the scheduling schemes is more pronounced. Although the two figures show different characteristics, the overall trend of the completion times is the same. First of all, the batch scheduling (PBS) has the lowest execution time, followed by the HYBRID scheme, and then the gang scheduling (SCore) and CC/SB, which have similar execution times (The performance results of CC, SB and that of the blocking scheme available with GM are almost identical and thus, we do not plot them separately.). The most striking observation in these figures is that the HYBRID coscheduling scheme has the lowest completion time among all scheduling schemes. The average completion time per job with the HYBRID scheme reduces as much as 68% compared to PBS and by 15% compared to the gang scheduler under heavy load.

The main reason the HYBRID scheme performs better than all prior coschedulings is that it avoids the unnecessary spinning time of other schemes. In CC and SB, a parallel process spins for a fixed time (e.g. 300us), and then is blocked if it is not complete. The weakness of this strategy is that the spinning time can be wasted if the corresponding process on the other node does not send the message in time. In SCore, DCS and PB, the communication is non-blocking and a process keeps spinning waiting for the message from another node. In contrast, in HYBRID, the process is immediately blocked if the communication operation is not complete. When the message arrives, the process wakes up immediately (since its priority is boosted) and starts communication with the remote node. Thus, the HYBRID scheme reduces the communication delay. The second reason is that there are frequent interrupts from a NIC to the CPU to boost the process's priority in CC, DCS and PB. In HYBRID, since the process is only boosted at the beginning of an MPI collective communication, it can save these interrupt overheads. Third, the HYBRID scheme avoids the global synchronization overhead of gang scheduling. A gang scheduler like SCore should spend some time to achieve the global synchronization across all the nodes per every scheduling quantum. However, since HYBRID follows the implicit coscheduling policy, there is no need for global synchronization.

Another interesting result is that the blocking-based coschedulings (HYBRID, CC and SB) have better performance than the non-blocking based coschedulings (DCS and PB). This is because the blocking technique allows other processes in the ready state to proceed and this improves the completion time as well as the throughput. Especially, the blocking-based mechanisms significantly outperform the PB scheme, contrary to some earlier results [32, 42, 5]. In [32] and [42], PB was shown as the best performer on a Solaris cluster through measurements and by an analytical model. In another simulation study [5], it was observed that simple blocking techniques were more effective in some cases than the spin-based techniques. However, our results indicate that blocking-based mechanisms (HYBRID and CC/SB) consistently perform better across all workloads in our cluster.

The main reason for this is the choice of the local scheduling algorithm. The earnings-based Linux scheduler uses a simple, *counter* controlled mechanism to restrict the amount of CPU time a process receives (within a bounded time, defined by an *epoch*). This ensures fairness to all processes in the same *class* [10]. On the other hand, priority-based, Multi-level Feedback Queue (MFQ) schedulers (as in Solaris, Windows NT) use a priority decay mechanism [40], where a process, after receiving a time slot, is placed in a lower priority queue and hence, cannot receive CPU time unless it bubbles up to the highest level. Thus, the amount of time a process receives is not as *tightly* controlled as it is in Linux. This means that if a process is made to remain in the highest level priority queue (which is the case in [32] for PB), it is difficult to ensure fairness. In contrast, in Linux, even if a temporary boost is given (to the PB scheme) on a message arrival, this boost cannot be continuously effective. This is because once the process uses its share of time slots, it can no longer be scheduled until all other processes also expire their time slots in the current *epoch*.

We conducted another set of experiments using BlockSolve95 [24], sPPM [26] and LU from the NAS benchmark by varying the problem size and the number of processors. These are relatively long-running applications with low communication. We generated 20 jobs mixed from these applications

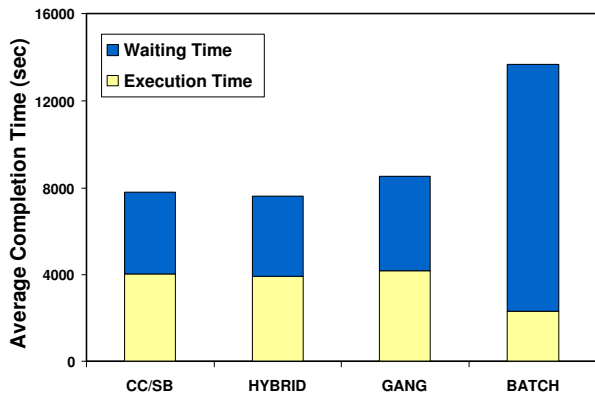


Figure 4: Completion Time Comparison with Long-running Applications (20 jobs)

with an average inter-arrival time of 500 seconds and MPL was limited to three. Figure 4 shows the completion time of PBS, SCORE, CC/SB and HYBRID schemes. Again, HYBRID exhibits the best performance followed by CC/SB, gang and PBS. Even for these low communication applications, batch scheduling suffers from the largest waiting time.

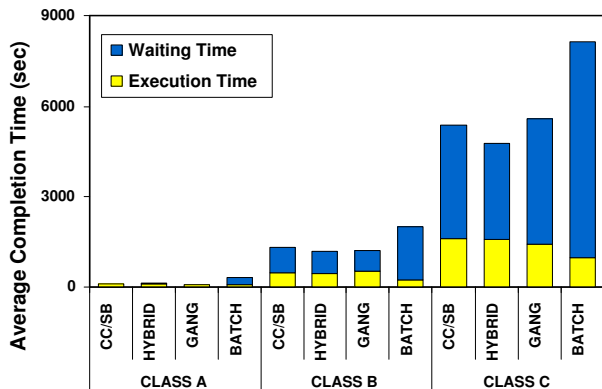


Figure 5: Performance Comparison with Different Problem Sizes (20 jobs)

Next, we analyze the scalability of the coscheduling techniques with various problem sizes. We use three largest classes of NAS workloads; A, B and C, and randomly generate 20 jobs in each CLASS with varying number of required processors for a job. In this experiment, we only test five scheduling schemes (CC, SB, HYBRID, PBS and SCORE), because HYBRID and CC always outperform PB and DCS techniques. Figure 5 shows the average completion time of the three workloads for different coscheduling schemes. In CLASS A, the completion times are very close, although the result of the batch scheduling shows higher waiting time. Since the execution times in CLASS A are too short, whenever a new job arrives at a cluster, it can start almost immediately. It makes the MPL as low as one. In CLASS B, batch scheduling (PBS) shows the worst performance due to maximum queuing time, despite its shortest execution time. In CLASS C, all four schemes show more noticeable results,

since the problem sizes are large enough to affect the performance. The completion time of the HYBRID and CC/SB schemes show about 10% improvement compared to SCORE. Comparing the result of HYBRID with that of PBS, we get at least 40% improvement.

Now, we focus on the performance variation when the communication intensities of the workloads change. In all 16 nodes, we simultaneously run three identical programs from CLASS C. In this experiment, we only consider the communication-driven coscheduling techniques and the gang scheduling because MPL in batch scheduling is always one. Since the three applications start and end almost at the same time, there is no waiting time. Therefore, we use average execution time as the performance metric for this experiment. In Figures 6 (a), (b) and (c), the HYBRID scheme always shows the shortest execution time for all NAS benchmarks. We omit the EP results here because the average execution times of all schemes are almost the same due to minimal communication in the EP application. The observation in this experiment reconfirms the fact that blocking-based coschedulings are better than spin-based coschedulings.

The NAS benchmarks in Figures 6 (a), (b) and (c) are grouped to show the difference properly. The trends of the results in Figure 6 are related to the communication intensities. CG, which has the highest communication intensity, shows the most distinguishable results between the blocking-based and the spin-based coschedulings. Although IS is also a high communication intensity program, its execution is too small to show noticeable difference between the scheduling policies. MG and SP, which are medium communication intensity programs, also show significant difference between the two groups of coscheduling schemes, while the execution time difference between the two groups for BT and LU (low communication intensity) is minimal.

### 5.3 Memory-Aware Allocation

In the previous experiments, we restricted the MPL to 3 to fit all the programs in the memory for gang scheduling and all coscheduling schemes. In a more general setting, the programs may compete with each other for the system memory if the total memory requirement exceeds the system memory size. To maximize the performance of parallel jobs, it is obvious that the programs should be memory resident to avoid memory swappings. This is the main reason why most dedicated clusters use a batch scheduler since they run large memory resident jobs and avoid expensive disk activities [2]. Some researchers [8, 38] have considered memory constraints only for a gang scheduler, but to the best of our knowledge, no previous study has been done for the communication-driven coscheduling. Therefore, in this section we analyze the performance implications when the parallel jobs are allocated considering the memory requirements.

Table 3 shows the maximum memory requirements of the NAS benchmarks, obtained using a system monitoring tool. In all NAS benchmarks, these values do not change during execution. Since the maximum MPL is three in our experiments, if the total memory requirement of three simultaneous running programs is larger than the system memory size, memory swapping is triggered. In this experiment, we use a workload consisting of randomly generated 20 jobs

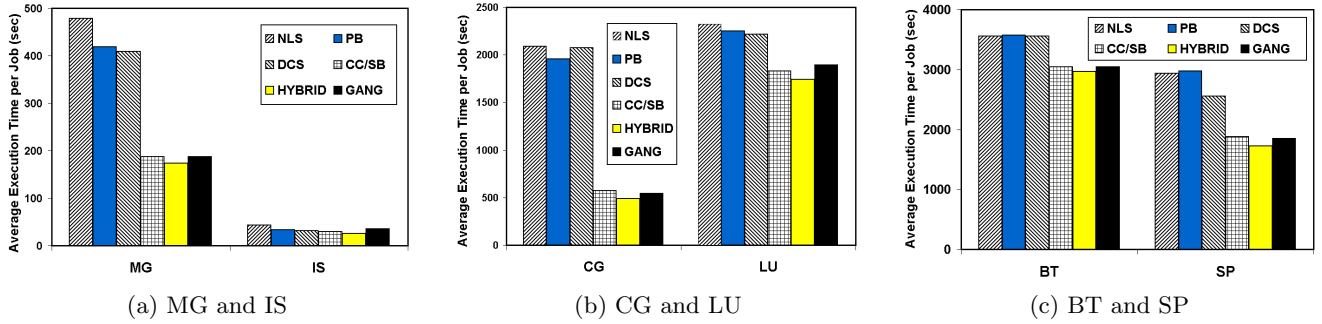


Figure 6: Average Execution Time for Different CLASSES of Applications

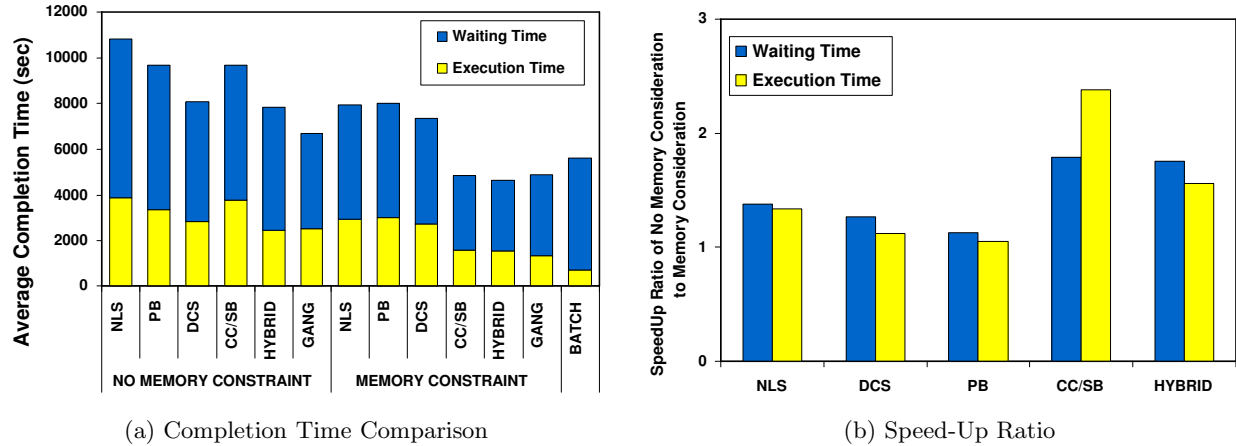


Figure 7: Completion Time Comparison with Memory-Aware Allocation (20 jobs)

NAS Benchmark Applications	Number of Processors	Maximum Memory Requirement (MBytes)
SP	4	354
SP	9	176
SP	16	111
BT	9	473
BT	16	272
MG	8	456
MG	16	228
IS	4	428
IS	8	214
IS	16	107

Table 3: Maximum Memory Requirements of NAS Benchmarks (CLASS=C)

shown in Table 3. These 20 jobs are generated with an average inter-arrival time of 100 seconds. We consider two scenarios; memory-aware and no memory-aware allocation. The no memory-aware allocation scheme does not check the memory requirements. Thus, the maximum MPL is always three. In the memory-aware allocation scheme, the allocator considers both the memory and CPU availability using the pre-determined memory information in Table 3. Execution of a job is delayed if the memory requirement of the job ex-

ceeds the available memory. So, the maximum MPL can be less than three. In this experiment, the maximum memory size allocated for the parallel programs is 900 MBytes, and the remaining 100 MBytes is reserved for the Linux Operating System (OS). We implement a memory-aware allocation module for the allocator used in the previous section. For the gang scheduling, we use the allocation modules provided by SCore.

Figure 7 (a) depicts the average completion time with different scheduling schemes. This figure includes just one result for the batch scheduling (PBS), because its MPL is always one. The first observation is that PBS outperforms all scheduling schemes with no memory-constraint. This is because the memory size of any NAS benchmark used in this experiment is smaller than the system memory. Therefore, no memory swapping occurs in batch scheduling. The average performance gain with the memory-aware allocation scheme is over 28% compared to the no memory-aware allocation. Again, the HYBRID scheme shows the best performance and its completion time is 17% lower compared to the batch scheduling. Note that the performance difference between PBS and HYBRID is not as high as in the previous graphs because here, only 20 jobs are tested for this experiment instead of 100 jobs due to long execution time of CLASS C applications. The difference will be large if 100 jobs are tested. Moreover, the MPL is less than 3 due to memory-constraints. The coscheduling schemes would per-

form much better at higher workloads with high MPL and when there is no memory swapping.

The speed-up ratio of no memory-aware allocation to memory-aware allocation is presented in Figure 7 (b). The execution time and waiting time speedup ratios of CC/SB and HYBRID are the highest, and the results imply that further performance gain is possible using a memory-aware allocator for the blocking-based schemes. Usually, memory swapping occurs when a process consumes its time quantum, but if a blocking mechanism is used, a process is likely to be context-switched out before completing its time quantum. Therefore, a blocking scheme may induce more memory swappings if the programs are not memory resident.

The main conclusion of this experiment is that a communication-driven coscheduling scheme should deploy a memory-aware allocator to avoid expensive disk activities. With this, it can outperform a batch scheduler.

## 5.4 Power Usage Comparison

While achievable performance is still the main issue in high performance systems, energy conservation is also becoming a serious concern because of the enormous amount of power required to maintain such infrastructures. As stated in the introduction, the operational power budget for maintaining a supercomputer may run into millions of dollars [27, 7, 25]. Thus, any viable approach for energy-efficient computing should be considered seriously. In this section, we examine the impact of all scheduling schemes, discussed previously, on the overall energy consumption in our 16-node cluster.

We use a power measurement device WT210 [47] in this experiment to measure power consumption of 6 nodes per unit time. We collect the average power consumption every 5 seconds from six randomly chosen nodes and then project the overall energy consumption of the whole cluster. Our cluster does not have the functionality of Advanced Power Management (APM) [21] or Advanced Configuration and Power Interface (ACPI) [22], which provides energy saving techniques in a machine. A single node in the cluster consumes about 219.10 watts when it is idle, and we refer to this as the *idle power consumption*. When a node is busy, it consumes an additional 18.839 watts, termed as *busy power consumption* in this paper.

Table 4 presents the total energy consumption of the different schemes in completing a workload of 100 mixed jobs from the NAS benchmarks. This is the same workload that was used to plot Figure 3 (a) in Subsection 5.2. In Table 4, the total energy consumption consists of two parts; *idle* energy and *busy* energy. The table reveals that the batch scheduling (PBS) has the highest energy consumption due to its longest waiting time, which translates to the corresponding idle energy. We observe that in addition to being able to provide better performance, the blocking-based coschedulings are quite energy-efficient. The HYBRID and CC/SB schemes exhibit 38% and 35% better energy saving compared to the batch scheduling, due to shorter completion times. Also, the gang scheduler is equally energy-efficient like the CC and SB schemes.

In addition, we measured the average utilization of individ-

Allocation Schemes	Scheduling Schemes	Energy Consumption (MJoule)		Node Utilization (%)	Total Running Time (sec)
		Busy	Idle		
NO_PACKING	NLS	9.40	86.63	96	24710
	PB	9.25	86.52	96	24679
	DCS	8.63	81.90	96	23361
	CC/SB	4.24	65.56	94	18700
	HYBRID	4.98	62.88	94	17938
PACKING	NLS	8.60	81.32	91	23198
	PB	8.20	79.42	92	22655
	DCS	7.66	74.69	92	21305
	CC/SB	4.56	58.02	91	16549
	HYBRID	4.82	56.38	91	16083
GANG		2.99	61.42	95	17520
BATCH		4.35	94.55	63	26971

**Table 4: Total Running Time, Utilization and Energy Consumption of the Scheduling Techniques (100 Jobs)**

ual nodes of the cluster during the execution of the 100 jobs. This results show that all scheduling schemes except PBS keep the node utilization at least 90%. Using this average node utilization, we can use the following simple equation to estimate the total energy consumption of any scheduling scheme:

$$E = (B + X \times U) \times N \times T, \text{ where}$$

- $E$  : total energy consumption
- $B$  : idle power consumption per node
- $X$  : busy power consumption per node
- $U$  : average node utilization
- $N$  : the number of nodes in a cluster
- $T$  : total program running time.

B and X are system dependent and in our case, B and X are 219.10 watts and 18.968 watts, respectively. Using the above equation, we calculated the energy consumption of all scheduling schemes and compared these values with the measured energy, shown in Table 4. The difference between the measured data and from the equation is within 3% across all the techniques. This analysis implies that we can get an accurate estimate of energy consumption if we know the system/node utilization and program execution time.

Figure 8 shows the total energy consumption trend of 20 jobs as a function of problem size. Obviously, the energy efficiency due to an efficient scheduling scheme increases as the problem size increases from CLASS A to CLASS C. With the long running CLASS C applications, the HYBRID scheme showed the best energy behavior followed by CC/SB and gang scheduling. Compared to the batch scheduler, the total energy consumption with these coscheduling schemes will decrease as the system load (in terms of number of jobs) increases.

Finally, Table 5 depicts the energy consumption of the memory-

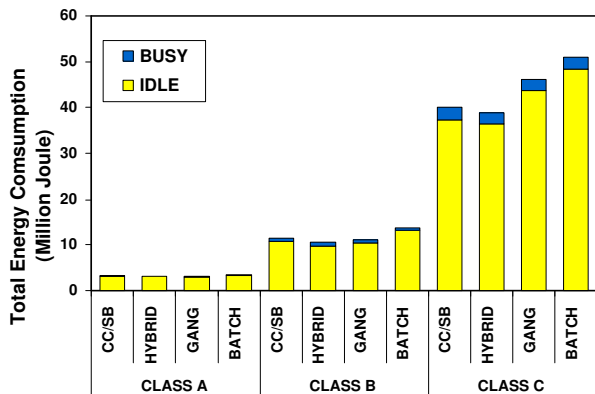


Figure 8: Variation of Energy Consumption with Problem Size

aware allocation experiment, which was discussed in Section 5.3. These results along with that of Figure 7 (a) indicate that by avoiding the disk swapping activities, we benefit both in terms of performance and energy.

Allocation Schemes	Scheduling Schemes	Energy Consumption (MJoule)		Node Utilization (%)	Total Running Time (sec)
		Busy	Idle		
NO MEMORY-AWARE	NLS	9.20	91.68	97	26151
	PB	7.84	79.09	92	22559
	DCS	6.92	69.05	91	19695
	CC/SB	4.73	89.55	87	25545
	HYBRID	4.16	59.94	90	17098
	GANG	6.19	61.27	99	17477
MEMORY-AWARE	NLS	7.59	74.19	97	21164
	PB	7.09	70.44	66	21551
	DCS	7.57	75.55	81	20092
	CC/SB	3.99	45.59	79	13003
	HYBRID	3.84	43.90	78	12522
	GANG	4.24	50.85	94	14505
BATCH		5.12	51.62	99	14724

Table 5: Total Running Time, Utilization and Measured Energy Consumption of Memory Allocation Schemes (20 Jobs)

The implications of this energy analysis study are the following: First, let us assume that a typical dedicated computer center like in LLNL or in any university computing facility runs some N number of jobs in a day using a batch scheduler like PBS [1] or LoadLeveler [19]. If we use the proposed HYBRID scheme or any other blocking-based coscheduling scheme (for that matter a gang scheduler), we can complete those jobs in much less time, implying that the system can be virtually shut down or can be in a power saving mode. One can argue that most centers probably would not shut down their system. In that case, using one of these efficient scheduling techniques, we can improve the system throughput, and thus, will achieve better performance-energy behavior. Supercomputers like the ASCI system are usually heavily loaded with many waiting jobs. An efficient

scheduler will certainly help in this scenario. Second, since the completion time difference between the batch scheduling and the three coscheduling schemes (HYBRID, CC and SB) is large, even we can use more sophisticated energy saving technique like dynamic voltage scaling (DVS) [11] at each node. This would save energy without degrading performance compared to a batch scheduler. Finally, we can completely shut down (or use to a sleep mode) a subset of the cluster nodes and can still run the applications without performance penalty using an efficient coscheduling scheme. Looking into these alternatives is essential for energy-aware computing.

## 6. CONCLUSIONS

The main conclusions of this paper are the following: First, although most clusters use variations of batch processing, we can get significant performance improvement by switching to a coscheduling mechanism like HYBRID, SB or CC. Our results on the 16-node platform showed up to 68% reduction in the average completion time with the proposed HYBRID scheme. We should get this type of improvement also on large clusters as long as the system is relatively loaded with mixed workloads. A gang scheduling technique like Score, which is currently used in several installations, can also provide better performance than a batch scheduler. However, the proposed HYBRID scheme outperformed it in all the experiments. Second, contrary to some previous results [32, 42], we recommend using blocking-based scheduling techniques (HYBRID, SB and CC) instead of the spin-only based techniques (DCS and PB) since the former schemes consistently outperformed the later techniques. Third, the proposed HYBRID scheme is the best performer and can be implemented on any platform with only modification in the message passing layer. SB is also a feasible technique that needs little effort for implementation. In view of this, the proposed framework is quite useful and can be used to implement more sophisticated techniques like CC for providing more customized scheduling. Fourth, like batch scheduling, any new technique deployed on a cluster should avoid expensive memory swapping. Finally, the improved efficiency of a scheduling algorithm translates to better performance-energy ratio, which could be used to explore several avenues to reduce the high maintenance cost. All these performance and energy results strongly support the case for using coscheduling algorithms in dedicated clusters.

We plan to expand our research in several directions. We would like to implement our framework on large and different platforms like Gigabit Ethernet [17], Quadrics [35] and IBA [20]. This would require implementing the framework in the Ethernet device driver and other NICs. Then, we would conduct measurements on large platforms using LLNL workloads. Moreover, we would like to investigate the performance-energy trade-offs more carefully by using Dynamic Voltage Scaling (DVS) and sleep mode for the cluster nodes.

## 7. REFERENCES

- [1] *OpenPBS*. Available from <http://www.openpbs.org>.
- [2] A. Acharya and S. Setia. Availability and Utility of Idle Memory in Workstation Clusters. In *Proc. of ACM SIGMETRICS'99*, pages 35–46, June 1999.

- [3] S. Agarwal, G. Choi, C. R. Das, A. B. Yoo, and S. Nagar. Co-ordinated Coscheduling in time-sharing Clusters through a Generic Framework. In *Proceedings of International Conference on Cluster Computing*, December 2003.
- [4] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.
- [5] C. Anglano. A Comparative Evaluation of Implicit Coscheduling Strategies for Networks of Workstations. In *Proceedings of 9th International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 221–228, August 2000.
- [6] A. C. Arpaci-Duseau, D. E. Culler, and A. M. Mainwaring. Scheduling With Implicit Information in Distributed Systems. In *Proceedings of the 1998 ACM SIGMETRICS joint International Conference on Measurement and Modeling of Computer Systems*, pages 233–243, June 1998.
- [7] A. M. Bailey. Accelerated Strategic Computing Initiative (ASCI) : Driving the Need for the Terascale Simulation Facility (TSF). In *Proceedings of Energy2002 Workshop and Exposition*, June 2002.
- [8] A. Batat and D. G. Feitelson. Gang Scheduling with Memory Considerations. In *Proceedings in 14th International Parallel and Distributed Processing Symposium*, pages 109–114, May 2000.
- [9] N. J. Boden et al. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
- [10] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., October 2000.
- [11] T. D. Burd and R. W. Brodersen. Design Issues for Dynamic Voltage Scaling. In *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 9–14, July 2000.
- [12] Compag, Intel and Microsoft. Specification for the Virtual Interface Architecture. Available from <http://www.viarch.org>, 1997.
- [13] T. V. Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface of Parallel and Distributed Computing. In *Proc. of 15th SOSP*, pages 40–53, Dec 1995.
- [14] Y. Etsion and D. G. Feitelson. User-Level Communication in a System with Gang Scheduling. In *In Proceedings of the International Parallel and Distributed Processing Symposium*, 2001.
- [15] D. G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report Research Report RC 19790(87657), IBM T. J. Watson Research Center, October 1994.
- [16] D. G. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, 23(5):65–77, May 1990.
- [17] Gigabit Ethernet Alliance. 10 Gigabit Ethernet Technology Overview White Paper. Available from <http://www.10gea.org/Tech-whitepapers.htm>.
- [18] A. Hori, H. Tezuka, and Y. Ishikawa. Highly Efficient Gang Scheduling Implementation. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–14, 1998.
- [19] IBM Corporation. *IBM LoadLeveler*. Available from <http://www.mppmu.mpg.de/computing/AIXuser/loadl>.
- [20] InfiniBand Trade Association. InfiniBand Architecture Specification, Volume 1 & 2, Release 1.1, November 2002. Available from <http://www.infinibandta.org>.
- [21] Intel and Microsoft. Advanced Power Management v. 1.2. Available from <http://www.microsoft.com/>.
- [22] Intel, Microsoft and Toshiba. The Advanced Configuration & Power Interface Specification. Available from <http://www.acpi.info>.
- [23] M. A. Jette. Performance Characteristics of Gang Scheduling in Multiprogrammed Environments. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–12, November 1997.
- [24] M. T. Jones and P. E. Plassmann. Solution of Large, Sparse Systems of Linear Equations in Massively Parallel Applications. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 551–560, November 1992.
- [25] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. A Comparison Between the Earth Simulator and Alphaserver Systems Using Predictive Application Performance Models. In *Proceeding of the International Parallel and Distributed Processing Symposium 2003*, pages 64–73, April 2003.
- [26] Lawrence Livermore National Laboratory. The sPPM Benchmark Code. Available from <http://www.llnl.gov/asci/purple/benchmarks/limited/sppm>.
- [27] Lawrence Berkeley National Laboratory. Data Center Energy Benchmarking Case Study, July 2003. Available from [http://datacenters.lbl.gov/docs/Data\\_Center\\_Facility4.pdf](http://datacenters.lbl.gov/docs/Data_Center_Facility4.pdf).
- [28] Lawrence Livermore National Laboratory. Accelerated Strategic Computing Initiative (ASCI) Program. Available from <http://www.llnl.gov/asci>.
- [29] B. Lawson, E. Smirni, and D. Puiu. Self-adapting Backfilling Scheduling for Parallel Systems. In *In Proceedings of the 2002 International Conference on Parallel Processing (ICPP 2002)*, pages 583–592, August 2002.
- [30] Myrinet, Inc. MPICH-GM software, October 2003. Available from <http://www.myrinet.com/>.
- [31] Myrinet, Inc. Myrinet GM-1 software, October 2003. Available from <http://www.myrinet.com/>.
- [32] 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.
- [33] NASA Advanced Supercomputing division. *The NAS Parallel Benchmarks (tech report and source code)*. Available from <http://www.nas.nasa.gov/Software/NPB/>.
- [34] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 55, December 1995.
- [35] Quadrics Ltd. QsNet HIGH PERFORMANCE INTERCONNECT. Available from <http://doc.quadrics.com/quadrics/Quadrics-Home.nsf/DisplayPages/Homepage>.
- [36] A. Rubini and J. Corbet. *Linux Device Drivers, 2nd Edition*. O'Reilly & Associates, Inc., June 2001.
- [37] H. P. Scott Rhine, MSL. Loadable Scheduler Modules on Linux White Paper. Available from <http://resourcemanagement.unixsolutions.hp.com>.
- [38] S. Setia, M. S. Squillante, and V. K. Naik. The Impact of Job Memory Requirements on Gang-Scheduling Performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(4):30–39, 1999.

- [39] S. K. Setia, M. S. Squillante, and S. K. Tripathi. Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems. *IEEE Trans. Parallel & Distributed Syst.*, 5(4):401–420, April 1994.
- [40] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 6th Edition*. John Wiley & Sons, 2001.
- [41] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. In *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pages 231–256, March 1998.
- [42] M. S. Squillante, Y. Zhang, A. Sivasubramaniam, N. Gautam, H. Franke, and J. Moreira. Modeling and Analysis of Dynamic Coscheduling in Parallel and Distributed Environments. In *Proc. of SIGMETRICS2002*, pages 43–54, June 2002.
- [43] Supercluster Research and Development Group. Maui Scheduler. Available from <http://supercluster.org/maui/>.
- [44] T. Takahashi, S. Sumimoto, A. Hori, H. Harada, and Y. Ishikawa. PM2: A High Performance Communication Middleware for Heterogeneous Network Environments. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 16, November 2000.
- [45] TOP500.org. TOP500 SUPERCOMPUTER SITES. Available from <http://www.top500.org>.
- [46] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.
- [47] Yokogawa Electric Cooperation. *WT210/WT230 Digital Power Meter USER'S MANUAL*, May 1998. Available from <http://www.yokogawa.com/>.
- [48] A. B. Yoo and M. A. Jette. The Characteristics of Workload on ASCI Blue-Pacific at Lawrence Livermore National Laboratory. In *Proc. of CCGrid2001*, pages 295–302, May 2001.
- [49] D. Zotkin and P. Keleher. Job-Length Estimation and Performance in Backfilling Schedulers. In *Proceedings of 8th International Symposium on High Performance Distributed Computing (HPDC'8)*, 1999.