Impact of Exploiting Load Imbalance on Coscheduling in Workstation Clusters

Jung-Lok Yu† Driss Azougagh† Jin-Soo Kim‡ Seung-Ryoul Maeng†
Division of Computer Science, Department of EECS
Korea Advanced Institute of Science and Technology (KAIST), South Korea
†{jlyu,driss,maeng}@calab.kaist.ac.kr ‡jinsoo@cs.kaist.ac.kr

Abstract

Implicit coscheduling is known to be an effective technique to improve the performance of parallel workloads in time-sharing clusters. However, implicit coscheduling still does not take into consideration the system behavior like load imbalance that severely affects cluster utilization. In this paper, we propose the use of global information to enhance the existing implicit coscheduling schemes. We also introduce a novel coscheduling approach - named PROC (Process ReOrdering-based Coscheduling) - based on process reordering exploiting global load imbalance information to coordinate communicating processes. The results obtained from an in-depth simulation study show that our approach significantly outperforms previous ones (by up to 38.4%) by reducing the idle time (by up to 86.9%) and spin time (by up to 36.2%) caused by the load imbalance.

1 Introduction

Workstation clusters are emerging as a platform for the execution of general-purpose workloads [1, 2]. For the successful use of clusters in domains such as scientific applications, databases, web servers and multimedia, etc., scheduling techniques are required that can effectively handle workloads with diverse demanding characteristics [1, 2, 4, 6, 15]. For those workloads, time-sharing approaches are particulary attractive because they provide good response times for interactive jobs and good throughput for I/O-intensive jobs. Unfortunately, time-sharing can be very inefficient for running parallel jobs that need process synchronization due to the lack of coordination among local schedulers [13, 15].

Over the years, two main strategies to coordinate individual local schedulers have been proposed in the literature: *explicit coscheduling* [3, 5] and *implicit coscheduling* [1, 7, 6, 13]. Explicit coscheduling [3, 5] uses explicit global knowledge constructed a *priori* and performs simultaneous global context switch to coschedule parallel processes across all CPUs. While it has been shown to be essential for fine-grained parallel applications, explicit coscheduling doesn't seem to be viable option for a cluster environment in that it suffers from high global synchronization overhead. Recently, another class of coscheduling schemes such as Demand-based Coscheduling (DCS) [7], Spin Block (SB) [6, 13] and Periodic Boost (PB) [1] have been proposed for cluster systems. These implicit coscheduling schemes use communication events - message arrival and response time - of parallel processes to guide the local schedulers toward coscheduled execution whenever needed. For example, on a message arrival (or fast response), the implication is that the sender (or corresponding) process is currently scheduled. Therefore, it will benefit to schedule, or keep scheduled the receiver process. Compared to explicit coscheduling, these schemes are easier to implement on cluster environments, and have better scalability and reliability.

From the above discussion, we raise two important questions. (i) *how optimal previous implicit coscheduling schemes are in terms of performance?* (ii) *if not, what are the missing factors that limit the system utilization?* We observe that most implicit coscheduling schemes rely only on the locally available information (message arrival and response time). We also realize that there is crucial global information, for example, load imbalance, representing the behavior of the system, which can be exploited to optimize the system utilization.

We argue that global load imbalance information are critical to implicit coscheduling in a cluster. Load imbalance is one of the major factors to interfere with the efficient utilization of clusters. Load imbalance has three main sources: 1) uneven load (computation, I/O, and communication) distribution to equally powerful computing nodes, 2) heterogeneity in cluster hardware resources, and 3) the presence of the local jobs and background (or daemon) jobs (multiprogramming) [4]. Since this load imbalance results in the increment of the idle time on CPU resources and the waiting time on communicating processes, it has a marked detrimental effect on cluster utilization. Therefore, exploiting

Proceedings of the 2005 International Conference on Parallel Processing (ICPP'05)
0190-3918/05 $20.00 © 2005 **IEEE**

globalized load imbalance can be the key point to implicit coscheduling to improve the performance of cluster. This paper presents a novel coscheduling approach that exploits both local and global information to answer above questions. At the best of our knowledge, no previous study has exhaustively investigated this issue in the context of implicit coscheduling on a cluster environment.

In view of this, we present an innovative coscheduling scheme, called **Process ReOrdering-based Coscheduling (PROC)**, based on process reordering which exploits global runtime information as well as the limited knowledge available locally to coordinate the communicating processes across all CPUs. We realize that the combination of the average CPU time spent by each process and the expected number of processes ready to be executed before the current process is rescheduled, represents the global load imbalance (and synchronization) information in the system. PROC measures these values dynamically at run-time, and exchanges the information by piggybacking them with normal messages. Based on the load imbalance information, the local scheduler can then make better coscheduling decisions by reordering processes with pending messages. Through an in-depth simulation study, we show that our approach significantly outperforms previous implicit coscheduling schemes by reducing the idle time and the spinning time caused by the load imbalance, thus improves cluster utilization.

The rest of the paper is organized as follows. In Section 2, we present the overview of the implicit coscheduling strategies proposed in the literature. Section 3 discusses the proposed PROC approach in details. Section 4 describes the simulation methodology and Section 5 discusses the results obtained from our experiments. Finally, Section 6 concludes the paper.

2 Related Work

As described in [13], implicit coscheduling are classified by two components: *message waiting action* taken by processes waiting for a message and *message handling action* performed by the kernel when a message arrives, as summarized in Table 1.

LOCAL is the most straightforward coscheduling technique. A receiving process is just spinning until the message arrives, and becomes coscheduled with the sender process only if the message arrives while it is spinning.

The next straightforward one is **Immediate Block (IB)**. In IB, the process blocks immediately if the message has not arrived yet, and is waken up by the kernel when the message eventually arrives. **Spin Block (SB)** [6, 13] is a compromise between LOCAL and IB. Here a process spins on a message arrival for a fixed amount of time, as referred to *spin time*, before blocking itself (called *two-phased spin blocking*). The underlying rationale is that a process waiting for a message should receive it within the spin time if the sender process is also currently scheduled. Consequently, if the message arrives within the spin time, the receiver process should hold onto the CPU to be coscheduled with the sender process. Otherwise, it should block in order not to waste the CPU resource. On subsequent message arrival, the network interface cards (NIC) raises an interrupt, which is serviced by the kernel to wake up the process and give a priority boost to the awaken process. As a variant of SB, Agarwal et al. proposed **Co-ordinated Coscheduling (CC)** [16], which performs sender-side optimization to coschedule parallel jobs. In the CC scheme, a sender spins for a fixed amount of time to wait for a send complete event. If a send is not completed within this time, it is implicitly inferred that the outstanding message queue at the NIC is long and hence, it is better to block and let another process use the CPU. However, these blocking-based schemes (IB, SB and CC) still have the limitation that they can not eliminate or reduce the idle time caused by load imbalance.

Table 1. Implicit coscheduling schemes

Scheme	Msg. Waiting Action		Msg. Handling Action
	Sender	Receiver	
LOCAL	Spin-Only	Spin-Only	Nothing
IB	Spin-Only	Imme. Block	Interrupt & Boost
SB (CC)	Spin-Only(-Block)	Spin-Block	Interrupt & Boost
DCS	Spin-Only	Spin-Only	Interrupt & Boost
PB	Spin-Only	Spin-Only	Periodic Boost

Demand-based CoScheduling (DCS) [7] uses an incoming message as an indication that the sending process is currently scheduled on the sender node. In DCS, a receiving process performs busy-waiting. Periodically, NIC finds out which process is currently running on its host CPU. On message arrival, the NIC checks whether the message destination process is currently executing or not. If there is a mismatch, an interrupt is raised. The interrupt service routine (ISR) boosts the priority of the destination process to coschedule it with the sending process. **Periodic Boost (PB)**, proposed in [1, 15], is an alternative coscheduling scheme to avoid expensive interrupt cost. In PB, the receiving process is busy-waiting like DCS. However, in this scheme, rather than raising an interrupt for each incoming message, a periodically invoked kernel thread examines message queues of each process, and boosts the priority of a process with pending messages based on some selection criteria. Whenever the scheduler is invoked in the near future, it would preempt the current process and schedule the boosted process. Obviously, these spinning-based schemes (DCS and PB) suffer from the time wasted by processes while spinning for messages to arrive. This problem can become more harmful when processes are highly imbalanced

in the cluster.

From the above description, we realize that the exploitation of global information (like ready-queue size in remote nodes) might solve most of those limitations. To exploit the global information, a new novel coscheduling scheme to efficiently reduce the idle time and the spinning time, is required. In this research, we introduce a coscheduling scheme based on reordering technique as an example to prove the importance of exploiting global load imbalance information.

3 Proposed Coscheduling Scheme

Exploiting the global load imbalance information has a major impact on the performance of cluster systems. Figure 1 shows the example of scheduling sequence performed by priority boost (and preemption) in each node. Note that normally, there are multiple incoming messages destined to distinct processes during a single scheduling quantum, and as a result of it, multiple processes are boosted (or waken up) until the next context switch. In this figure, we assume that N_1 and N_3 are the nodes with low load, and N_2 is heavily loaded (load(N_1) < load(N_3) < load(N_2)). In spinning-based schemes, N_1 and N_3 suffer from the spinning time if N_2 schedules the processes with pending messages without considering the load status of N_1 and N_3 (see Fig. 1(a.1)). As depicted in Fig. 1(a.2), P_1 and P_3 can be scheduled in advance in N_2 if N_2 realized that N_1 and N_3 have the lower load than other remote nodes. Similarly, in blocking-based schemes, N_1 and N_3 suffer from the idle time if N_2 schedules the awaken processes regardless of the loads of N_1 and N_3 (see Fig. 1(b.1)). As shown in Fig. 1(b.2), using load imbalance information from N_1 and N_3, N_2 can schedule P_1 and P_3 at time t and $t^{'}$ to reduce the idle time in N_1 and N_3. Therefore, by scheduling in advance a process whose corresponding processes will be scheduled sooner in remote nodes, we are able to decrease the spinning time and the idle time. This allows parallel processes to achieve better progress.

As described above, although the load imbalance has a marked detrimental effect on cluster's utilization, most implicit coscheduling strategies described in Section 2 take no account of the load imbalance to coordinate communicating processes. To address this concern, **PROC (Process ReOrdering-based Coscheduling)** measures the load imbalance information dynamically at run-time, and exchanges the information by piggybacking them with normal messages. Based on the load imbalance information, the local scheduler can then make better coscheduling decisions by reordering processes with pending message(s).

Then, the next question is how to measure the load imbalance information. In fact, it is very difficult to correctly measure the degree of load imbalance with a little overhead.

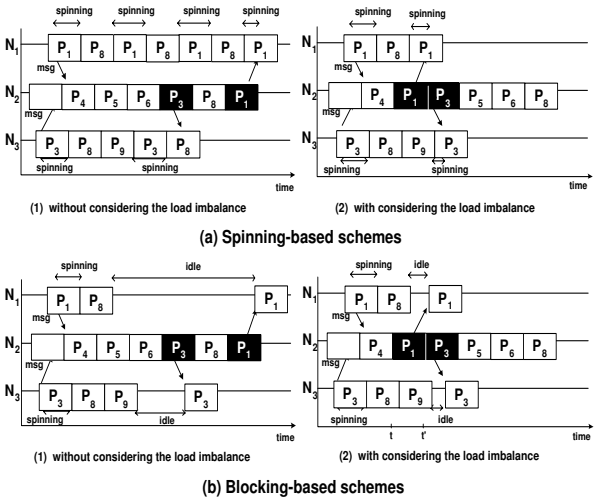


Figure 1. Effect of load imbalance in (a) spinning-based schemes and (b) blocking-based schemes

In order to minimize the overhead for measuring the degree of load imbalance, we use a heuristic algorithm as follows.

At any time, each node has a current process that uses a CPU. Each node N_i can compute: (a) the average CPU time spent by each process (averaged time difference between consecutive context switches) (TS_i), and (b) the expected number of processes ready to be executed (ENP_i) before the current process is scheduled again. In this paper, ENP_i is calculated to the summation of: (1) the number of processes with ready-to-run state (or with pending messages) in the highest-level ready queue on N_i and (2) the average number of processes to be additionally waken(or boosted) up by I/O completion and message arrival during the time interval ($TS_i \times$ the number of processes obtained from (1)).

Let us assume that there is a system with *N* nodes where each node N_i contains P processes. Each node N_i piggybacks TS_i and ENP_i in every outgoing messages as the load imbalance information of N_i. When a process P_k in N_j receives a message from N_i at time *t*, we define the followings:

- $\boldsymbol{TS_{ijk}} \leftarrow TS_i, \boldsymbol{ENP_{ijk}} \leftarrow ENP_i$
- $\boldsymbol{T_{ijk}}$: the latest time a process P_k in N_j receives a message from N_i ($T_{ijk} \leftarrow t$)
- $\boldsymbol{T_{ij}}$: the time of the last received message by N_j from N_i ($T_{ij} = \max_k(T_{ijk})$)
- $\boldsymbol{TS_{ij}}$: the most recent TS_i of N_i received by N_j

Each time N_j receives a new message from N_i, NIC updates a data structure (in scheduling layer) related to

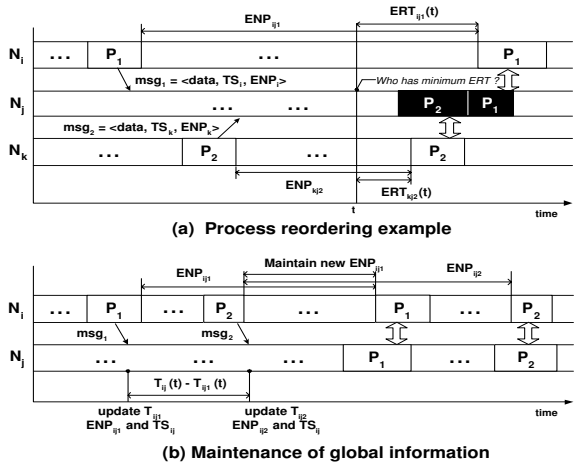


Figure 2. Process reordering example and maintenance of global information

the load imbalance information (ENP_{ijk}, T_{ijk}, TS_{ij}, and T_{ij}) of the remote node N_i based on TS_i and ENP_i extracted from the message. As each process with pending message(s) contains a list of the most recent load imbalance information of remote nodes, our reordering algorithm makes a new order among processes in N_j by sorting local processes mainly based on the Expected Remaining Time (ERT) to schedule the corresponding processes in remote nodes (see Fig. 2(a)). Our reordering algorithm is shown in the Algorithm 1.

The ERT_{ijk} represents the expected remaining time to schedule the corresponding process of local process P_k in the remote node N_i. It is updated by extracting the time spent in N_j from the total expected remaining time required to reschedule the corresponding process in N_i, as shown in line 14 in the algorithm. In line 15, we determine ERT_{jk} which represents the minimum ERT_{ijk} among all remote nodes. Based on the ERT_{jk}, our reordering algorithm computes the least Expected Reordering Factor (ERF) in N_j (ERF_j) among all processes with pending message(s). The Candidate Set of the Preferable processes (CSP) in N_j contains all processes with ERT_{jk} equal to the least ERF_j (see from line 18 to 22 in the algorithm). It represents the set of the most urgent processes which should be scheduled first. Note that when the queue has less than two processes with pending messages, this reordering procedure is not invoked. For the simplicity reason, the scheduler randomly selects one candidate process from the set CSP to be scheduled next. Before computing the CSP, processes receiving messages from the same remote node, they can share load imbalance information and maintain their information as shown from line 9 to 12 in the algorithm (see also Fig. 2(b)).

Algorithm 1: Process reordering algorithm

```
1  Reordering Procedure (node N_j, current time t, CSP) {
2    CSP = null;
3    ERF_j = infinite;
4    for each process P_k with pending message(s) in N_j {
5      ERT_jk = infinite;
6      for each message m of P_k {
7        i = sender node of message m;
8        // maintain the load imbalance information
9        if ( T_ijk < T_ij ) {
10         ENP_ijk = ENP_ijk - ((T_ij - T_ijk) / TS_ij);
11         T_ijk = T_ij;
12       }
13       // determine minimum ERT value in a process
14       ERT_ijk = (TS_ij * ENP_ijk) - (t - T_ijk);
15       if ( ERT_jk > ERT_ijk ) ERT_jk = ERT_ijk;
16     }
17     // determine a process set with minimum ERF value
18     if ( ERF_j > ERT_jk ) {
19       ERF_j = ERT_jk;
20       CSP = { k };
21     }
22     else if ( ERF_j == ERT_jk ) CSP = CSP + { k };
23   }
24 }
```

For experimental purpose, SB and PB are selected as two case studies since they represent the most successful and rich strategies among others. In SB, our reordering algorithm can be applied at each scheduler invocation by determining the process with minimum ERT value. In contrast to SB, PB needs to apply the reordering algorithm at each PB kernel thread invocation ($\sim$ 1ms). For the convenience, we call the former case **SB+PROC**, and the latter **PB+PROC**.

4 Experimental Methodology

Simulator. We used a detailed, process-oriented event-driven simulator, named ClusterSchedSim [17] built on CSIM19 [19] package, and added our proposed scheme onto it. As depicted in Fig. 3(a), each workstation com-

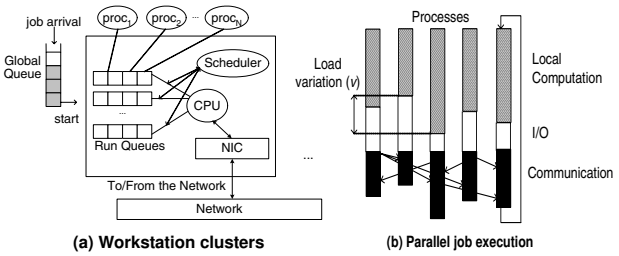


Figure 3. Simulation model of workstation cluster and parallel job execution

Table 2. Workloads characteristics

	Comp.	I/O	Comm.	*WL1*	a set of J1
J1	70	5	25	*WL2*	a set of J2
J2	48	5	47	*WL3*	a set of J3
J3	25	5	70	*WL4*	equal mix of J1,J2,J3

(a) Synthetic Workloads

	Pattern	Comm.	Msg. Size (bytes)
LU	NN (four)	11.5%	320, 640 (48.3%) 40,960, 81,920 (1.56%)
CG	NN (six)	63.4%	8 (59%), 16 (1%) 14,000(40%)
IS	AA, Barrier	30.1%	4 (50%) 32,768 (50%)
FT	AA	56.3%	128KB (50%) 256KB (50%)

(b) Realistic Workloads

Table 3. Simulation parameters and values

Parameters	Value(s)
System size	32
MPL(Multi-Programming Level)	5, 10
Communication patterns	NN, AA
Message size	32 KB
One-way latency	187.97 μs
Variance (v)	0.5, 1.5
Context switching cost	100 μs
Interrupt processing cost	30 μs
Check an endpoint	2 μs
Download (or upload) of global info.	2 μs
Change the position in scheduling queue	2 μs

prises a NIC, OS scheduler, and a set of user processes. The NIC module models the interactions between user processes (or scheduler) and the network. Whenever a message is received from the network, the NIC delivers it into a user buffer and raises an interrupt. Similarly, the NIC waits for outgoing messages and enqueues them into the network module. This form of operation is typical of user-level communication approach [8]. Costs for these operations have been obtained from microbenchmarks performed on a cluster of Pentium III-800 MHz workstations connected by Myrinet [9]. The scheduler module emulates Solaris scheduler [10] and is responsible for manipulating a priority-based multi-level feedback queue (60 queues) on which ready-to-run processes are placed. Each workstation may run an arbitrary number of user processes, whose executions are expressed by a simple language that allows the specification of computations, disk I/O and communication operations. For the global scheduler, we adopt FIFO.

The periodic boost mechanism used in PB and PB+PROC becomes active every one millisecond. For SB and SB+PROC, we set the *spin time* for a message to be the expected one-way latency. In both SB+PROC and PB+PROC, costs for downloading (or uploading) the global information to NIC (or to scheduling layer), calculating and comparing the ERT values, and changing the position in the scheduling queue are modeled in the simulator.

Workloads. We consider two types of workloads: synthetic and realistic. Synthetic workloads are generated from San Diego Supercomputer Center (SDSC) SP2 traces, which are widely used in scheduling studies [11, 12]. During the synthetic workload generation, job arrival time, execution time, and size information are characterized to fit a mathematical model called Hyper-Erlang distribution of common order [14]. Each job in the workload requires 32 processes and iterates phases of local computation, disk I/O, and interprocess communication. We consider two different communication patterns: Nearest Neighbor (NN) and All-to-All (AA), which are commonly used in many parallel scientific applications. We assume that both communication patterns use a fixed message size of 32KB. By fixing the end-to-end one-way latency of a message, the computation and I/O time per iteration can be calculated. By multiplying the computation and I/O time by a value uniformly selected in (1 + *unif*(-v/2, v/2)) and by varying the load variance (v), we model the load imbalance across CPUs (see Fig. 3(b)).

In order to obtain realistic workloads, four parallel applications (LU, IS, CG, and FT) have been directly derived from the NAS Parallel Benchmarks (NPB) 2.4 suite [18]. More specifically, these applications have been obtained by translating their source codes in NPB into the language accepted by the ClusterSchedSim, without changing their execution flow, communication topology, and message sizes. The duration of sequential parts of these parallel application codes have been determined from measurements performed by running the corresponding NPB applications on a cluster of Pentium III-800MHz workstations. The characteristics of workloads and the simulation parameters used in our experiments are summarized in Table 2 and Table 3, respectively.

5 Experimental Results

5.1 Benefit Analysis of PROC

Here, we examine the results concerning the impact of workload characteristics (different proportions of computation, I/O, and communication) and communication patterns on the performance of different coscheduling schemes. *WL1*, *WL2*, and *WL3* represent computation-intensive, well-balanced, and communication-intensive workload, respectively. For this experiment, we limit the maximum Multi-Programming Level (MPL) to five and set the load variance factor (v) to 0.5. Figure 4 shows the average job response

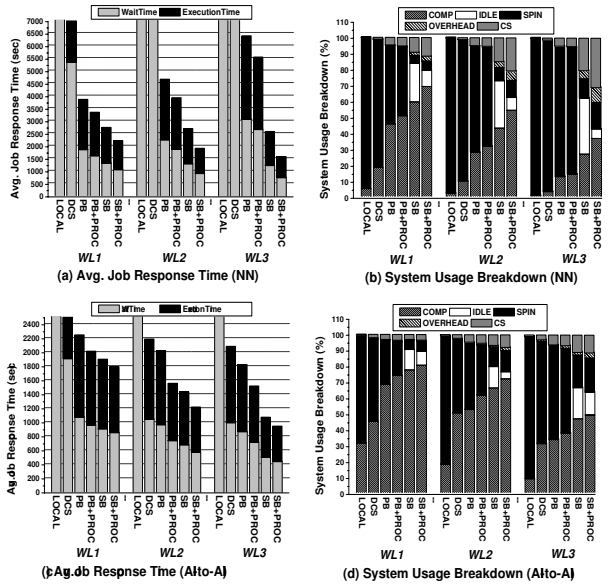
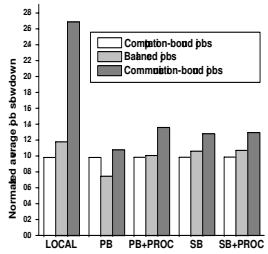


Figure 4. Impact of workload characteristics and communication patterns (MPL = 5, v = 0.5)

time and the system usage breakdown of our interesting six coscheduling schemes for these three workloads with Nearest Neighbor and All-to-All communication patterns.

The most striking observation in these figures is that the proposed coscheduling schemes, PB+PROC and SB+PROC, achieve better performance than PB and SB, respectively. We also note that SB+PROC has the lowest average job response time among all scheduling schemes. PB+PROC scheme reduces the average job response time by up to 23.1% compared to PB, and SB+PROC scheme by up to 38.4% compared to SB. The main reason the reordering-based schemes outperform prior coschedulings is that they avoid the unnecessary spinning time or idle time of previous schemes occurred by load imbalance. In DCS and SB, the boost sequence of a parallel process is determined by message arrival, and in PB by simple round-robin fashion. In contrast, PB+PROC and SB+PROC reorder the boost sequence of a parallel process according to the global load imbalance (by boosting a process with minimum ERT as described in Section 3), trying to reduce the unnecessarily wasted time. From Fig. 4(b) and 4(d), we can see that PB+PROC reduces the spinning time of PB by up to 36.2% and SB+PROC reduces the idle time of SB by up to 86.9%. When reordering is applied, the overhead is increased due to the cost for the maintenance and appliance of load imbalance information described in Section 3, and the context switch is increased since the process reordering makes the probability of scheduling appropriate (or urgent) processes high. However, these additional costs do little affect the overall benefit. Across all workloads, we also find that the blocking-based schemes (SB and SB+PROC) show better performance than the spinning-based schemes (LOCAL, DCS, PB and PB+PROC). This is because blocking technique allows processes of other applications to proceed in their computations, thus improving the response time.



(a) Normalized avg. job slowdown

scheme	coeff. of variation
LOCAL	0.456
PB	0.157
PB+PROC	0.165
SB	0.107
SB+PROC	0.111

(b) Coefficient of variation of slowdown

Figure 5. Fairness (*WL4* with NN, MPL=5, v=0.0)

Next, to evaluate the fairness, we calculate the coefficient of variation of slowdown over three different types of jobs in *WL4* with NN[1] (see Fig. 5). As depicted in Fig. 5, our scheme has almost the same fairness value as previous schemes. We also notice that blocking-based schemes are more fair than spinning-based schemes.

5.2 Effect of Load Imbalance and Multi-Programming Level (MPL)

In this section, we examine the effect of the load imbalance and Multi-Programming Level (MPL) on the performance of the considered different coscheduling approaches. In this experiment, we exclude LOCAL and DCS because there is no point to show their performance. We consider two extreme scenarios that have less communication (*WL1* with NN communication pattern) and intensive communication (*WL3* with AA) to analyze the behavior of reordering in relation with the load variance and MPL.

Figure 6 shows the average job response time and the system usage breakdown for these workloads with two different load variance values (0.5 and 1.5) and two different MPL values (5 and 10). All obtained results show that even with highly imbalanced load (or with larger MPL), applying reordering (PB+PROC and SB+PROC) enhances the performance of the previous schemes.

In Fig. 6 (right two groups of bars of each graph), since the higher load imbalance makes the probability of mismatch of communicating processes high, we observe that the response time increases with a larger load vari-

[1]The results for *WL4* with AA are omitted due to space limitation.

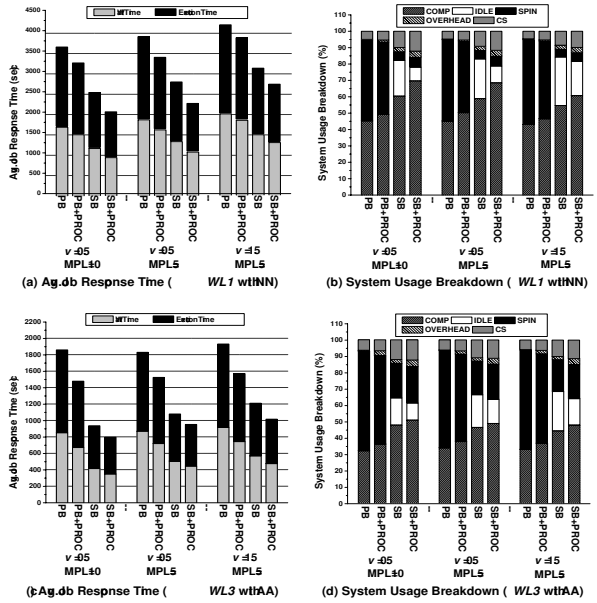


Figure 6. Impact of load imbalance and MPL

ance value. This increment is mainly affected by the spinning time increase for the spinning-based approaches (PB and PB+PROC) and the idle time increase for the blocking-based approaches (SB and SB+PROC). From Fig. 6(a) and 6(c), we know that the effect of load imbalance (the increment of job response time) can be better hidden in AA than in NN due to the overlapping between communication and computation.

5.3 Realistic Workloads Performance

Finally, we consider four realistic workloads (CG, IS, LU, and FT) as described in Table 2(b). In each experiment, all jobs require 32 nodes, and are started simultaneously. Figure 7 reports the slowdown of implicit coscheduling schemes relative to BATCH (estimated as the ratio of the last job completion time divided by the sum of the execution times of the all applications run in isolation) and the system usage breakdown for these four realistic workloads.

In Fig. 7, it is clearly shown that with the use of global load imbalance information and reordering, PB+PROC and SB+PROC significantly outperform PB and SB, respectively, for all realistic workloads. Again, SB+PROC performs the best across all realistic workloads, and consistently shows speedup compared to BATCH. It is observed that for the application with low communication intensity like LU, there is hardly any need for coscheduling (note that in LU case, spinning-based schemes perform even worse than LOCAL). However, as the communication intensity

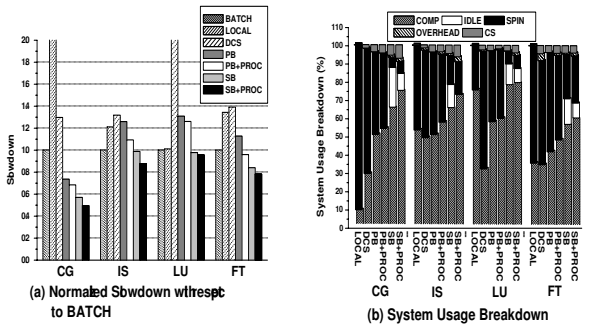


Figure 7. Realistic workloads performance (MPL = 5)

increases (LU < IS < FT < CG), the impact of implicit coscheduling becomes prominent. In CG, which has the highest communication intensity, PB+PROC shows about 30% speedup compared to BATCH, and SB+PROC gets at least 50% speedup. Also, the observation in this experiment reconfirms the fact that blocking-based schemes perform better than spinning-based schemes.

5.4 Discussion

From the previous results, applying the reordering mechanism substantially enhances the performance of PB and SB. Using the global load imbalance information, our reordering scheme tries to avoid the unnecessary spinning time or idle time of previous schemes occurred by load imbalance. This makes the exploitation of global load imbalance information as a main key point for our reordering scheme as well as any future coming reordering variants in clusters. Accordingly, in this section, we introduce the results of average message pending time for more analysis.

Table 4 shows the average message pending time (MSG_PENDING_TIME) of previous and proposed schemes for NN communication pattern [2] with different communication intensity. MSG_PENDI-NG_TIME is defined by the average time spent when messages arrive until they are consumed. Since implicit coscheduling schemes record small MSG_PENDING_TIME compared to LOCAL, the employment of implicit coscheduling onto workstation clusters is strongly recommended. We also notice that a considerable MSG_PENDING_TIME difference between spinning-based schemes (DCS, PB and PB+PROC) and blocking-based schemes (SB and SB+PROC) proportionally reflects the performance achieved in terms of average job response time in all previous results. MSG_PENDING_TIME is reduced by up to

[2]The results with AA are omitted because the overall trend of MSG_PENDING_TIME with AA is the same as with NN.

Table 4. Average message pending time (NN, MPL = 5, v = 0.5)

	MSG_PENDING_TIME		
	WL1	*WL2*	*WL3*
LOCAL	94.590 msec	88.875 msec	98.494 msec
DCS	19.039	15.958	20.959
PB	6.909	5.912	5.916
PB+PROC	6.202	5.587	5.641
SB	4.762	2.913	2.115
SB+PROC	3.726	2.039	1.347

10% in PB+PROC compared to PB and 36% in SB+PROC compared to SB. This reduction represents one of the key point of our reordering scheme. Our reordering algorithm favorites urgent processes that have high expectation to achieve synchronization with their corresponding ones in remote nodes in the near future. This fact reduces the MSG_PENDING_TIME, and consequently allows a process in average to consume its messages quicker and proceed for further executions.

6 Conclusion and Future Work

In this paper, we proposed the use of global information to address the main limitation of existing implicit coscheduling schemes - less accurate decision on who to boost to be coscheduled without regard to the load imbalance. We also presented a novel coscheduling approach based on process reordering exploiting global load imbalance information to coordinate the local schedulers.

We used the synthetic and realistic workloads to evaluate PROC compared to other schemes. We performed various experiments to analyze how the exploitation of global information using our reordering technique impacts on the performance of implicit coscheduling. The results reported in this paper show that our approach clearly provides better performance by reducing the idle time and the spinning time, thus improving the utilization of clusters. In PB+PROC, we achieved the improvement in terms of average job response time by up to 23.1%, while in SB+PROC, by up to 38.4%.

We plan to explore more global information that affects the coordination among communicating processes such as message frequency, queue size in NIC, etc. We also plan to extend our work by considering sequential and interactive jobs, and to implement PROC in a Linux cluster.

References

[1] Y. Zhang, A. Sivasubramaniam et al., Impact of Workload and System Parameters on Next Generation Cluster Scheduling, *IEEE Transactions on Parallel and Distributed System*, 2001, 12-9, pp. 967-985

[2] C. Anglano, A Comparative Evaluation of Implicit Coscheduling Strategies for Networks of Workstations, High Performance Distributed Computing, 2000, pp. 221-228

[3] J. Ousterhouw, Scheduling techniques for concurrent systems, 3rd International Conference on Distributed Computing Systems, 1982, pp. 22-30

[4] U. Rencuzogullari and S. Dwarkadas, Dynamic adaptation to Available Resources for Parallel Computing in an Autonomous Network of Workstations, Principles and Practice of Parallel Programming, 2001, pp. 72-81

[5] D. Feitelson and M. Jette, Improved Utilization and Responsiveness with Gang Scheduling, Job Scheduling Strategies for Parallel Processing, 1997, pp. 238-261

[6] A. Dusseau, R. Arpaci and D. Culler, Effective Distributed Scheduling of Parallel Workloads, ACM SIGMETRICS Conf. MMCS, 1996, pp. 25-36

[7] P. Sobalvarro, S. Pakin, et al., Dynamic Coscheduling on Workstation Clusters, IPPS Workshop on JSSPP, 1998, pp. 231-256

[8] D. Dunning et al., The Virtual Interface Architecture, *IEEE Micro*, 1998, pp. 66-75

[9] N. Borden et al., Myrinet: A Gigabit-per-second Local Area Network, *IEEE Micro*, 1995, 15, pp. 29-36

[10] SUN Microsystems Inc., Solaris 2.6 Software Developer Collection, 1997, Available from http://www.sun.com/

[11] G. Sabin, R. Kettimuthu, et al., Scheduling of Parallel Jobs in a Heterogeneous Multi-Site Environment, JSSPP, 2003, pp. 87-104

[12] D. Feitelson, Metric and Workload Effects on Computer Systems Evaluation, *Computer*, 2003, pp. 18-25

[13] S. Nagar, A. Banerjee, et al., Alternatives to Coscheduling a Network of Workstations, *Journal of Parallel and Distributed Computing*, 1999, 59-2, pp. 302-327

[14] H. Franke, J. Jann, et al., Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific, Supercomuting, 1999.

[15] S. Nagar, A. Banerjee, et al., A Closer Look at Coscheduling Approaches for a Network of Workstations, ACM Symp. Parallel Algorithms and Architectures, 1999, pp. 96-105

[16] S. Agarwal, G. S. Choi, et al., Coordinated Coscheduling in Clusters through a Generic Framework, Cluster Computing, 2003, pp. 84-91

[17] Y. Zhang and A. Sivasubramaniam, ClusterSchedSim: A Unifying Simulation Framework for Cluster Scheduling Strategies, *SIMULATION: Transactions of the Society for Modeling and Simulation*, May 2004, pp. 191-206

[18] NAS division., The NAS parallel benchmarks Available from http://www.nas.nasa.gov/Software/NPB/

[19] H. D. Schwetman, CSIM19: a powerful tool for building system models, 2001 Winter Simulation Conference, 2001, pp. 250-255