PROC : Process ReOrdering-based Coscheduling on 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

Workstation clusters are emerging as a platform for the execution of general-purpose workloads. To use clusters as shared computing servers, scheduling techniques able to effectively handle workloads with diverse characteristics on demands, are required. 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 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 based on process reordering exploiting global load imbalance information to coordinate communicating processes. The results obtained from a detailed simulation study show that our approach significantly decreases the average job response time (by up to 21.4%) by reducing the idle time (by up to 55.6%) and spin time (by up to 31.8%) caused by the load imbalance.

1. Introduction

The advent of fast networks [1, 2, 3] and efficient user-level communication protocols [4, 5, 9] has made clusters an attractive alternative to traditional multiprocessor systems. Due to the incremental scalability, cost-effectiveness and high-availability, clusters are gaining acceptance not just in scientific applications that need supercomputing power, but also in domains such as databases, web servers and multimedia [6, 7].

To use clusters as general-purpose shared computing servers, scheduling techniques able to effectively handle workloads with diverse characteristics on demands, are required [6, 10, 15, 17]. For diverse workloads, time-sharing approaches are particulary attractive because they provide good response times for interactive jobs and good throughput for I/O-intensive jobs. However, time-sharing has the drawback that communicating processes must be scheduled simultaneously to obtain good performance. The lack of coordination among local schedulers prevents parallel processes from being simultaneously scheduled on the respective CPUs when they need to synchronize, resulting in the performance degradation of parallel applications. Therefore, some form of coordination among individual schedulers must be provided to achieve satisfactory performance for parallel applications.

The possible alternatives to coordinate individual local schedulers span from a naive local scheduling to more sophisticated approaches like explicit coscheduling (or gang scheduling) [8, 11] or implicit coscheduling (sometimes also called communication-driven coscheduling) [13, 12, 18, 26]. In local scheduling, each local scheduler schedules its own processes independently without any effort to coschedule them. On the other hand, explicit coscheduling [8, 11], uses explicit global knowledge constructed a *priori* and simultaneous global context switch to ideally coschedule parallel processes across all CPUs. However, it usually requires long time quanta to amortize the high context switch and synchronization costs, which results in poor interactions with interactive or I/O jobs. Furthermore, it keep the CPU idle while a process is doing I/O or waiting for a message within its allotted time quantum. Some variants of explicit coscheduing recently appeared in the literature [17, 15] solve some of above problems, but do not seem enough suitable and reliable to handle general-purpose workloads in a cluster environment.

Understanding the practical limitations in realizing explicit coscheduling for cluster systems, a myriad of implicit coscheduling schemes such as Demand-based Coscheduling (DCS) [13, 14], Spin Block (SB) [12, 18] and Periodic Boost (PB) [26], have been recently proposed to effectively schedule parallel jobs. These schemes use the communication behavior of parallel processes to make scheduling decisions aimed at achieving a satisfactory coscheduling degree. The implicit information available for implicit coscheduling consists of two inherent communication events: mes-

Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)
1530-2075/05 $ 20.00 **IEEE**

sage arrival and response time. When a message arrives, the implication is that the sending process is currently scheduled. Therefore, it will benefit to schedule, or keep scheduled the receiving process. Also, a fast response intimates to the sending node that the receiving process is currently scheduled. Therefore, the proper action is to keep the sender scheduled. Conversely, if the response is not received within a threshold, the sending node can infer that the receiving process might be not scheduled. Thus, it is not beneficial to keep the sender scheduled. As contrast with explicit coscheduling, these schemes are easier to implement on cluster environments, and have better scalability and reliability. In this paper, we limited our work on implicit coscheduling on non-dedicated workstation clusters.

From the above discussion, we raise two important questions. First, how much optimum are previous implicit coscheduling schemes in terms of performance? Second, 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 representing the behavior of the system, which can be exploited to optimize the system utilization. This paper presents a novel coscheduling approach that exploits both local and global information to answer above questions.

We argue that global load imbalance and synchronization 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) [10]. 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. Also, from described above, it is easy to induce that reducing the synchronization delay among communicating processes has a major impact on implicit coscheduling. Therefore, globalized load imbalance and synchronization information can be the key point to implicit coscheduling to improve the performance of cluster. 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 addition, we believe that the study, presented in this paper, may reveal new directions for future researches of implicit coscheduling.

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. The proposed PROC measures these values dynamically at run-time, and exchanges the information by piggybacking them with normal messages. Based on the load imbalance and synchronization information, the local scheduler can then make better coscheduling decisions by reordering processes with pending messages.

Through a detailed simulation study, our experiments reveal several significant results. First, blocking-based schemes like SB and SB+PROC perform significantly better than spinning-based schemes like DCS, PB and PB+PROC, which is similar to the results reported in [7, 16] but contrary to [6, 26]. Second, the above observation leads us to the conclusion that choice of the local scheduler and workloads has a significant impact on the coscheduling schemes. Third, we demonstrate that the proposed reordering-based approach outperforms other schemes over a range of different workloads in terms of the response time and overall system throughput without sacrificing fairness. This is because reordering communicating processes based on the global information reduces the idle time and the spinning time. Finally, we argue that reducing the time message spent in a node before it is consumed as well as increasing the synchronization ratio is important to provide improved performance of clusters.

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 and the relevant work. 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 [18], implicit coscheduling schemes are classified by two components: *message waiting action* taken by processes waiting for a message and *message handling action* performed by the operating system when a message arrives, and are 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 if the message arrives while it is spinning. The next straightforward one is **Immediate Block (IB)**. In IB, the process

Table 1: Implicit coscheduling schemes

Message Handling Actions	Message Waiting Actions			
	Spin	Block	Spin Block	Spin Yield
None	LOCAL	IB	SB (and CC)	SY
Interrupt & boost	DCS	IB-DCS	SB-DCS	SY-DCS
Periodically boost	PB	IB-PB	SB-PB	SY-PB

blocks immediately if the message has not arrived yet, and is waken by the kernel when the message eventually arrives.

Spin Block (SB) [12, 18] 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. To block itself, the process which does not receive a message within spin time, makes a system call, and this information is informed to network interface cards (NIC). On subsequent message arrival, the 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, Ararwal 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 schemes still can not eliminate or reduce the idle time caused by load imbalance due to the lack of global coordination.

Demand-based CoScheduling (DCS) [13, 14] 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 [26], is an alternative coscheduling scheme which addresses the inefficiency arising from 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

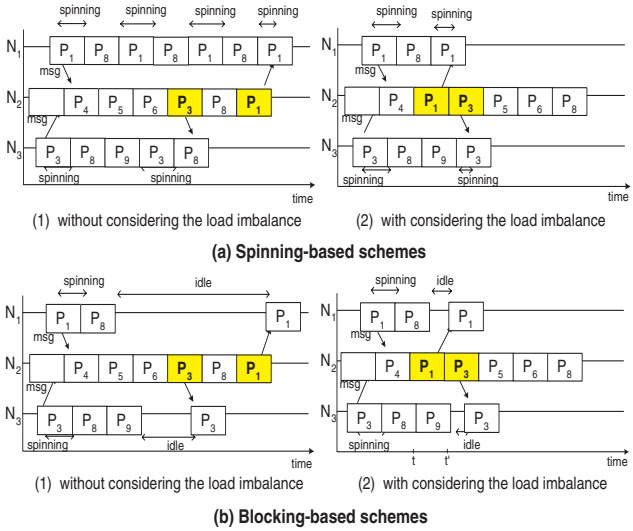


Figure 1: Effect of load imbalance in (a) spinning-based schemes and (b) blocking-based schemes

messages. Whenever the scheduler is invoked in the near future, it would preempt the current process and schedule the boosted process. There are several heuristics to decide on who to boost when the kernel thread is invoked. These heuristics take the local states of processes with pending message(s) into consideration in picking a candidate for a priority boost. 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 information.

3. Proposed Coscheduling Scheme

Exploiting the global load imbalance and synchronization information has a major impact on the performance of cluster systems. For instance, 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)) as shown in Fig. 1. In spinning-based schemes, N_1 and N_3 suffer from the spinning time if N_2 schedules the boosted processes 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 realize that N_1 and N_3 have the lower load than other remote nodes. Similarly, in blocking-based schemes, N_1 and

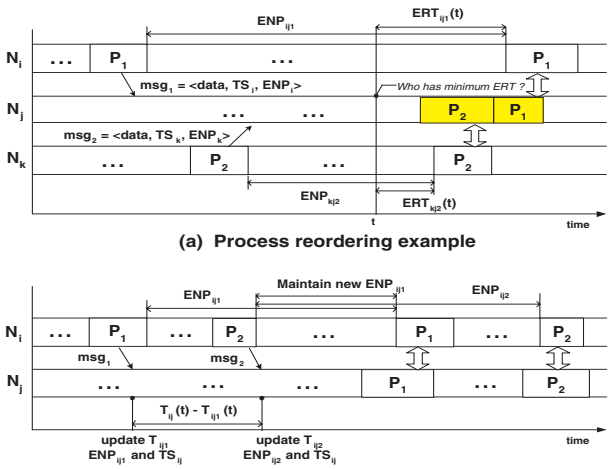


Figure 2: Process reordering example and maintenance of global information

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 (or in a time-fashion way) a process in which some of its corresponding processes in remote nodes will be scheduled sooner, we are able to decrease the spinning time and the idle time, and to achieve better progress among parallel processes.

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. It is due to the absence of global load imbalance and synchronization information in the system. To address the problem, we propose a novel coscheduling approach called **PROC (Process ReOrdering-based Coscheduling)**. To improve the overall system utilization, PROC 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).

At any time, each node has a current process that uses a CPU. Each node N_i can compute: (1) the average CPU time spent by each process (averaged time difference between consecutive context switches) (TS_i), and (2) 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 in the highest-level ready queue on N_i (in spinning-based schemes, the number of processes with pending messages on N_i is added to this value) and 2) the average number of processes to be additionally waken up by I/O completion and message arrival during the time interval ($TS_i \times$ the number of processes obtained from 1)).

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 }
```

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:

- ***TS*$_{ijk}$** and ***ENP*$_{ijk}$** : $TS_{ijk} \leftarrow TS_i$, $ENP_{ijk} \leftarrow ENP_i$
- ***T*$_{ijk}$** : the latest time a process P_k in N_j receives a message from N_i ($T_{ijk} \leftarrow t$)
- ***T*$_{ij}$** : the time of the last received message by N_j from N_i ($T_{ij} = \max_k(T_{ijk})$)
- ***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 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 a 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)).

For experimental purpose, **SB** and **PB** are selected as two case studies, since they represent the most successful and rich strategies among others to be taken in consideration. Our reordering algorithm can be applied in SB, at each scheduler invocation, by maintaining the *ENP*s and *T*s, reordering the processes in the highest-level ready queue based on the value ERT_{jk} at that time, and scheduling one of the candidates in the set CSP. In contrast to SB, PB need to apply the reordering algorithm at each PB mechanism (or a kernel thread) invocation ($\sim$ 1ms). For the convenience, we call the former case as **SB+PROC**, and the latter as **PB+PROC**.

4. Experimental Methodology

4.1. The Simulator

We used a detailed, process-oriented event-driven simulator, named ClusterSchedSim [21] built on CSIM19 [22] simulation toolkit, and added our PROC scheme on it. As depicted in Fig. 3(a), the simulation model of each workstation comprises a NIC, OS scheduler, and application processes. Also, the simulator has a network module which connects the computing elements together. Since our work focuses mainly on implicit coscheduling, we use a simple linear model for the network which is parameterized by the message size, without considering network contention.

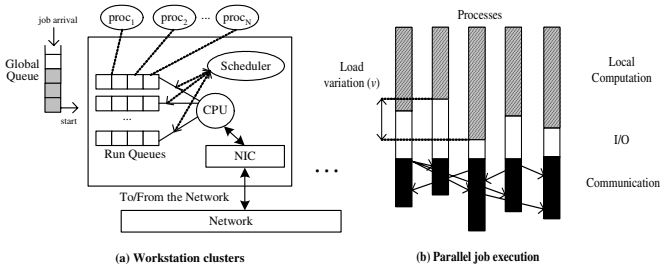


Figure 3: Simulation model of workstation cluster and parallel job execution

And, we adopt a global scheduler based on FIFO to schedule arrival jobs to the system (no process migration and no backfilling).

The NIC module models the interactions among the scheduler, application processes, and the network. Whenever a message is received from the network, the NIC delivers it into an application 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 [4, 5, 9, 23]. Costs for these operations have been obtained from microbenchmarks performed on a cluster of Pentium III-800 MHz workstations connected by Myrinet [1]. The scheduler module emulates Solaris scheduler [24] and is responsible for manipulating a 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.

There are two modules added on the scheduler, which are interrupt service routine (ISR) module and periodic boost module. ISR module is used for SB and SB+PROC which use blocking as the message waiting action. It is invoked immediately after the NIC module raises an interrupt. After considering interrupt processing costs, it manipulates the scheduling queue to boost the priority of the process to be woken. The periodic boost module is used for PB and PB+PROC, and is invoked periodically (every one millisecond). At each invocation, it examines the message queues of application processes and manipulates the scheduling queue to boost the priority of the process with pending messages.

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.

Table 2: Workloads characteristics

Job Characteristics			
Job Type	Comp.(%)	I/O(%)	Comm.(%)
J1	70	5	25
J2	40	20	40
J3	25	5	70
LU	comm. intensity: low		
FT	comm. intensity: high		

Workloads	
WL	Job types in Workload
WL1	a set of J1
WL2	a set of J2
WL3	a set of J3
WL4	equal mix of J1, J2 and J3
WL5	FT:FT:FT:FT:FT:FT
WL6	LU:J1(NN):J2(AA):J2(AA):FT:FT

Table 3: Simulation parameters and values

Parameters	Value(s)
System size	32
MPL(Multi-Programming Level)	5, 10
Communication patterns	NN, AA
Message size	4KBytes
One-way latency	143.40 μs
Variance (v)	0.5, 1.5
Context switching cost	200 μs
Interrupt processing cost	30 μs
Check an endpoint	2 μs
Download (or upload) of global info.	1 μs
Change the position in scheduling queue	2 μs

4.2. Workload Characterization

The real workloads of cluster systems can be characterized with the dynamic behaviors such as dynamic job arrival, different job size and execution time, and different job characteristics (computation, communication and I/O ratio), etc. In order to simulate those realistic workloads, we generate synthetic workloads derived from Cornell Theory Center (CTC) SP2 traces, which are widely used in scheduling studies [19, 20]. During the workload generation, job arrival time, execution time, and size information are characterized to fit a mathematical model called Hyper-Erlang distribution of common order [27]. The synthetic workloads used consist of 200 parallel jobs, where each job 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 and scientific applications. We assume that both communication patterns use a fixed message size of 4KB. By fixing the end-to-end one-way latency of a message, the communication cost per iteration in the ideal case is known. Based on this communication cost and the relative proportion of the other two phases (computation and I/O), the computation and I/O time per iteration can be calculated. By multiplying the computation and I/O time of each iteration of a process that were calculated earlier 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)).

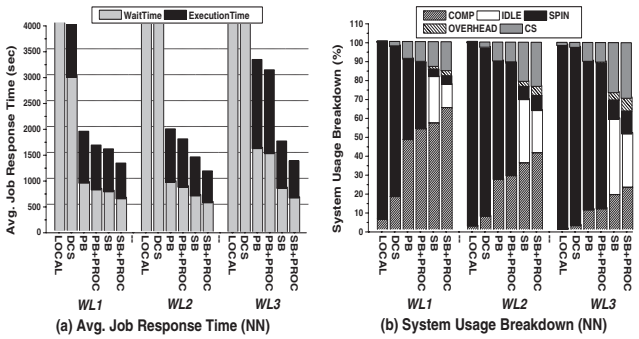


Figure 4: Impact of workload characteristics and communication patterns (NN, MPL = 5, v = 0.5)

Our workloads are completed by two parallel applications (LU and FT) that have been directly derived from the NAS Parallel Benchmarks (NPB) suite [25]. More specifically, these 2 applications have been obtained by translating their source codes in NPB into the language accepted by our simulator, without changing their execution flow, communication topology and message sizes. The choice of these 2 applications is based on their different computation granularity, communication intensity and patterns: LU is a lower/upper triangular matrix decomposition application with low communication intensity and NN communication pattern, while FT performs a multi-dimensional Fast Fourier Transform (FFT) with high communication and AA. 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

In this section, we present and discuss the results of our experiments. We consider the average job response time and/or completion time as performance metrics. Average job response time is defined by the time difference between the job completion and the job arrival averaged over all jobs. In addition, to better understand the performance results, we use system usage breakdowns showing the percentage of time that a CPU spends on average in different com-

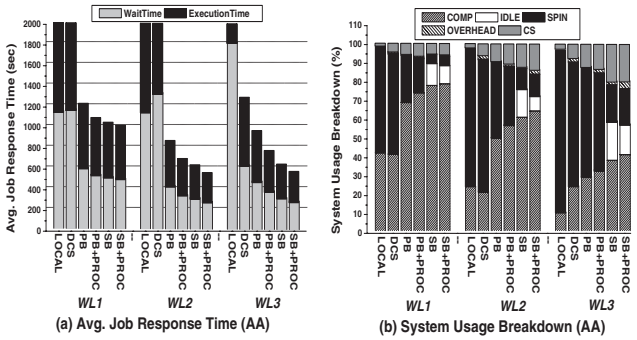
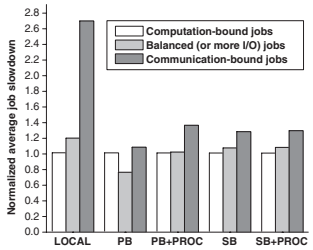


Figure 5: Impact of workload characteristics and communication patterns (AA, MPL = 5, v = 0.5)



(a) Normalized avg. job slowdown

Scheme	coeff. of var.
LOCAL	0.456
PB	0.157
PB+PROC	0.165
SB	0.107
SB+PROC	0.111

(b) Coefficient of variation of slowdown

Figure 6: Fairness (*WL4* with NN, MPL = 5, v = 0.0)

ponents such as compute, idle, spinning, context switches, and overheads (such as interrupt cost, reordering cost, etc.). Our experiments start by analyzing the performance of proposed PROC and the impact of various system parameters (like Multi-Programming Level and load imbalance) using the synthetic workloads (*WL1* ~ *WL4*). Then, we complete this section by analyzing the effect of mixed workloads on PROC using two realistic applications described in the previous section.

5.1. Benefit Analysis of Process Reordering-based Coscheduling

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* represents computation-intensive, well-balanced (with more I/O compared to *WL1* and *WL3*), and communication-intensive workload, respectively. Figure 4 and 5 show the average job response time and the system usage breakdown of our interesting six coscheduling schemes for these three workloads with Nearest Neighbor and All-to-All as the communication pattern, respectively. For this experiment, we fix the value of MPL (Multi-Programming Level) to 5 and load variance (v) to 0.5.

From Fig. 4 and 5, we can see that the reordering schemes achieve better performance than all other previous strategies. We also observe that the blocking-based schemes give more chance that processes of other applications make progress in their computation than the spinning-based schemes. This observation can explain that **SB+PROC** achieves better performance than **PB+PROC**.

In particular, we note that **PB+PROC** reduces the spinning time of **PB** by up to 31.8% and **SB+PROC** reduces the idle time of **SB** by up to 55.6%. 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. Also, the results exhibit that the performance of AA is better than that of NN. In Fig. 4, the response time is increased as the ratio of communication becomes high, while in Fig. 5, it is decreased as the communication ratio becomes high. This can be explained by the fact that the overlapping between communication messages in each iteration of AA is higher than that of NN.

In Fig. 6, to evaluate the fairness, we calculate the coefficient of variation of slowdown over three different classes of jobs in *WL4* (the results for *WL4* with AA are omitted due to space limit). As depicted in Fig. 6, 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.

As described in the above results, **PB+PROC** and **SB+PROC** use the load imbalance and synchronization information, reduce the spinning time and the idle time, respectively, and achieve better average job response time as we expect in Section 3. (see Fig. 1)

5.2. Effect of Load Imbalance

In this section, we examine the effect of the load imbalance 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 factor. For this experiment, we fix the value of MPL to 5. Figure 7 (right two groups of bars of each graph) 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). All obtained results show that with

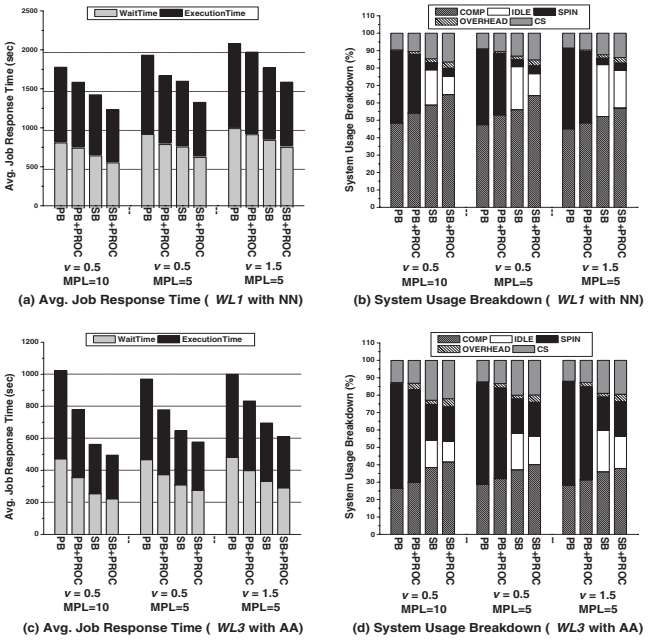


Figure 7: Impact of load imbalance and MPL

a larger load imbalance, applying reordering (PB+PROC and SB+PROC) enhanced the performance of the previous schemes by up to 16.7% and 10.4% for PB and SB, respectively.

In Fig. 7, 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 variance 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). Also, from Fig. 7(a) and 7(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. As a remark, the load imbalance has more impact on applications with computation-intensive than communication-intensive.

5.3. Effect of Multi-Programming Level (MPL)

Similarly to the previous section, we use the same experiment setup to analyze the effect of MPL on the performance when our reordering is applied. The response time and the system usage breakdown, when the load variance (v) value is fixed to 0.5, are shown in Fig. 7 (left two groups of bars of each graph). The results show performance improvement when applying reordering (PB+PROC and SB+PROC) compared to the previous schemes. With a larger MPL, 23.8% and 13.2% of improvements are achieved by PB+PROC and SB+PROC, respectively.

From the figure, when MPL increases, the job response time is decreased in general due to the large amount of useful work. As instance, all scheduling approaches in both communication patterns (NN and AA) achieve better job response time when MPL is equal to 10 than when MPL is equal to 5. The only exception of the above statement is the cases of PB and PB+PROC when *WL3* with AA is selected (see Fig. 7(c)). These two cases can be explained by the fact that, when communication intensive workload is selected, the probability of having larger spinning time among communicating processes and the context switches among local processes become high. This can be enforced by observing that in all cases the context switch increases with a larger MPL.

5.4. Mixed Workload Performance

Finally, we consider the realistic workload (*WL5*) and the mixed workload (*WL6*) with a combination of synthetic and realistic workloads, varying in computation granularity, communication intensity and patterns as described in Table 2. All jobs in *WL5* and *WL6* have been adjusted so they approximately take the same amount of time (13 ˜ 15 sec) to complete, when executed individually. For this experiment, MPL and load variance (v) values are fixed to 6 and 0.0, respectively. Fig. 8 shows the completion time and system usage breakdown of *WL5* and *WL6*. Table 4 shows SYNC_RATIO and MSG_PENDING_TIME of the results shown in Fig. 8 for *WL6*. SYNC_RATIO represents the ratio of the received messages when their corresponding processes are scheduled to the number of all messages. MSG_PENDING_TIME indicates the average pending time of messages spent in the system before their corresponding processes can consume them.

From the results, it is clearly shown that with the use of global load imbalance information and reordering, PROC outperforms previous schemes. The mixture of the workload including of realistic workloads and the separate running of FT, ensure the validity of our reordering scheme concerning the improvement. We exclude the results of LU due to the similarity to those achieved in *WL1* with NN pattern regarding the system usage breakdown and the improvement. Due to the space limit we omitted the analysis of the completion time and the system usage breakdown which can be easily induced from the above sections. Our focus, will be concerned the impact of the workload mixture on the performance and the analysis of the SYNC_RATIO and MSG_PENDING_TIME results.

The SYNC_RATIO shows a slight improvement when reordering technique is applied. This implies that the probability for a message to be consumed right when it just ar-

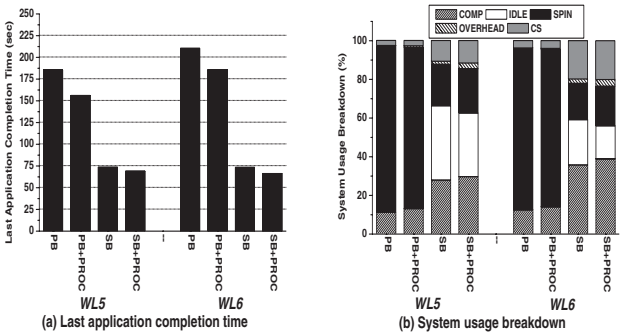


Figure 8: Mixed workload performance

Table 4: Synchronization ratio and average message pending time

	SYNC_RATIO	MSG_PENDING_TIME
PB	40 %	44.416 msec
PB+PROC	41 %	38.197 msec
SB	12 %	15.823 msec
SB+PROC	14 %	11.252 msec

rives mainly depends on the message waiting action adopted by the coscheduling scheme. As can be seen in PB in Table 4, the large value of SYNC_RATIO is due to the large amount of time consumed by processes while spinning for some messages. While in SB, the low value of SYNC_RATIO can be explained by the fact that once a processes can't find out the message it gets blocked.

Instead of SYNC_RATIO, MSG_PENDING_TIME is improved by 14% in PB+PROC compared to PB and 28% in SB+PROC compared to SB. This improvement 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 reduces the MSG_PENDING_TIME, and consequently allows a process in average to consume its messages quicker and proceed for further executions.

5.5. 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 increases the chance of synchronizing communicating processes and decrease the average message pending time. 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.

In computation-intensive workload, we observe that ERT is more accurate since TS value is high and has low variance. However, the low communication intensity degrades the importance of the global information. This degradation limits the harness of the improvement of proposed PROC. In contrast, as the amount of communications in each iteration becomes high, the size of CSP becomes large enough to allow more overlapping among communicating processes. This makes the global information (used in this paper) worthy enough for our reordering scheme to increase the degree of improvement. However, communication-intensive workload suffers from the relatively inaccurate ERT due to a small value of TS and its high variance. This inaccuracy saturates the improvement of our reordering scheme when the communication intensity exceeds some point as shown in Fig. 9.

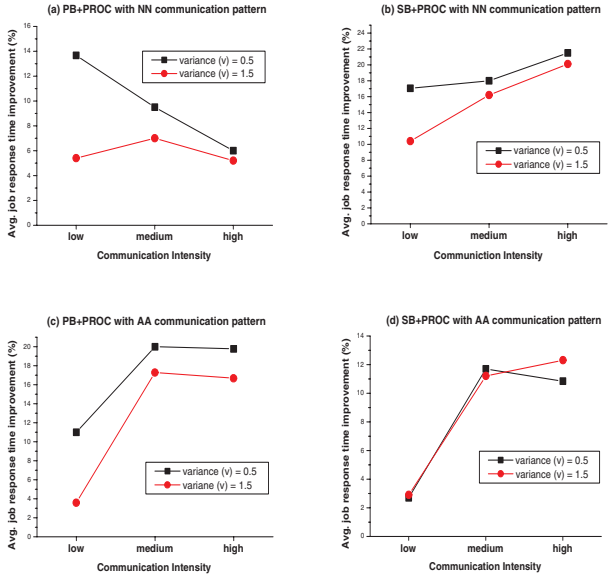


Figure 9: Average job response time improvement of PROC scheme

From the Fig. 9, in high load variance (v), the computation ratio for each iteration varies dramatically inducing the high variance of TS value and rather low accuracy of ERT. Thus, the low accuracy of ERT results in degrading the degree of improvement of our reordering approach. As a result, our reordering scheme needs to use more sophisticated metrics of load imbalance information for more ERT accuracy when communication-intensive workloads and high load variance (v) are applied to the system.

6. Conclusion and Future Work

In this paper, we proposed the use of global information to enhance the existing implicit coscheduling schemes. We also presented a novel coscheduling approach, named PROC (Process ReOrdering-based Coscheduling) based on process reordering exploiting global load imbalance infor-

mation to coordinate the communicating processes. Our approach addresses the main limitation of previous implicit coscheduling schemes - less accurate decision on who to boost to be coscheduled regardless to the load imbalance, increasing the chance of synchronization among communicating processes and decreasing the average message pending time.

We used the synthetic and realistic workloads with two different communication patterns to evaluate PROC compared to other schemes. We performed various experiments to analyze how the exploitation of global information using our reordering impacts on the performance of implicit coscheduling. The results reported in this paper show that our approach clearly provides better average job response time by reducing the idle time and the spinning time, thus improves the utilization of clusters. In PB+PROC, we achieved the improvement in terms of average job response time by up to 20%, while in SB+PROC, by up to 21.4%.

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 more real applications, including sequential and interactive jobs, and implement PROC in a Linux cluster.

References

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

[2] G. E. Alliance, "10 gigabit ethernet technology overview white paper", Available from http://www.10gea.org/Tech-whitepapers.htm

[3] Emulex Corp., "The VI/IP Standard and cLan", Available from http://www.emulex.com

[4] T. von Eicken, et al., "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", In Proc. 15th ACM Symp. on Operating System Principles, 1995, pp. 40-53

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

[6] Y. Zhang 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

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

[8] J.K. Ousterhouw, "Scheduling techniques for concurrent systems", In Proc. of the 3rd International Conference on Distributed Computing Systems, 1982, pp. 22-30

[9] J.S Kim, et al., "Design and Implementation of a User-level Sockets Layer over Virtual Interface Architecture", Concurrency and Computation: Practice and Experience, 2003, pp. 727-749

[10] 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

[11] D.G. Feitelson and M. A. Jette, "Improved Utilization and Responsiveness with Gang Scheduling", In Proc. of JSSPP, 1997, pp. 238-261

[12] A.C. Dusseau, et al., "Effective Distributed Scheduling of Parallel Workloads", In Proc. ACM SIGMETRICS 1996 Conf., 1996, pp. 25-36

[13] P.G. Sobalvarro, "Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors", PhD thesis, Massachusetts. Inst. of Technology, Jan. 1997

[14] P.G. Sobalvarro, et al., "Dynamic Coscheduling on Workstation Clusters", Proc. IPPS Workshop on JSSPP, 1998, pp. 231-256

[15] E. Frachtenberg, et al., "Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources", In IPDPS03, 2003, 15, 7-8, pp. 625-651

[16] S. Agarwal, G. S. Choi, et al., "Co-ordinated Coscheduling in Clusters through a Generic Framework", ACM SIGMETRICS Conference, June 15 - 19, 2002

[17] F. Petrini and Wu-chun Feng, "Improved Resource Utilization with Buffered Coscheduling", *Journal of Parallel Algorithms and Applications*, 2001, 16

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

[19] Dror G. Feitelson, "Metrics for Parallel Job Scheduling and their Convergence", In JSSPP, Dror G. Feitelson and Larry Rudolph, (ed.), *Springer Verlag, Lect. Notes Comput. Sci.* 2001, pp. 188-205

[20] G. Sabin, et al., "Scheduling of Parallel Jobs in a Heterogeneous Multi-Site Environment", In JSSPP, Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, (ed.), *Springer Verlag, Lect. Notes Comput. Sci.* 2003, pp. 87-104

[21] 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, vol. 80, no. 4-5, pp. 191-206

[22] H. D. Schwetman, "CSIM19: a powerful tool for building system models", In Proc. of the 2001 Winter Simulation Conference, 2001, pp. 250-255

[23] J. L. Yu, et al, "An Efficient Implementation of Virtual Interface Architecture using Adaptive Transfer Mechanism on Myrinet", In Proc. of ICPADS2001, 2001, pp. 741-747

[24] SUN Microsystems Inc., "Solaris 2.6 Software Developer Collection", 1997, Available form http://www.sum.com/

[25] N. A. S. division., "The NAS parallel benchmarks", Available from http://http://www.nas.nasa.gov/Software/NPB/

[26] S. Nagar, et al., "A Closer Look at Coscheduling Approaches for a Network of Workstations", In Proc. of 11th ACM Symp. Parallel Algorithms and Architectures, 1999, pp. 96-105

[27] H. Franke, et al., "Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific", In Proc. Supercomuting, 1999.