Impact of Job Allocation Strategies on Communication-Driven Coscheduling in Clusters*

Gyu Sang Choi[1], Saurabh Agarwal[2], Jin-Ha Kim[1], Anydy B. Yoo[3], and Chita R. Das[1]

[1] Department of Computer Science and Engineering
{gchoi,sagarwal,jikim,das}@cse.psu.edu
[2] IBM India Research Labs
New Delhi – 110016, India
{saurabh.agarwal}@in.ibm.com
[3] Lawrence Livermore National Laboratory
Livermore, CA 94551
{yoo2}@llnl.gov

Abstract. In this paper, we investigate the impact of three job allocation strategies on the performance of four coscheduling algorithms (SB, DCS, PB and CC) in a 16-node Linux cluster. The job allocation factors include Multi Programming Level (MPL), job placement, and communication intensity. The experimental results show that the blocking based coscheduling schemes (SB and CC) have better tolerance to different job allocation techniques compared to the spin based schemes (DCS and PB), and the local scheduling. The results strengthen the case for using blocking based coscheduling schemes in a cluster.

1 Introduction

Recently, several dynamic coscheduling algorithms have been proposed for improving the performance of parallel jobs on cluster platforms, which have gained wide acceptance from cost and performance standpoints [1][2][3]. These coscheduling algorithms rely on the communication behavior of the applications to schedule the communicating processes of a job simultaneously. Using efficient user-level communication mechanisms such as U-Net [4], Fast messages [5], and Virtual Interface Architecture (VIA) [6], these techniques are shown to be quite efficient. All prior coscheduling studies have primarily focuses on the scheduling of parallel processes assigned to a processor. They do not address the job allocation issue, *i.e.*, the mechanism to assign jobs to the required nodes of a cluster. Allocation is an integral part of a processor management technique and can have a significant impact on the overall system performance. In view of this,

* This research has been supported by NSF grants CCR-9900701, CCR-0098149, CCR-0208734, and EIA-0202007.

several job allocation strategies have been proposed for batch and gang scheduling techniques to improve system performance [7]. In this paper, we investigate the impact of several job allocation strategies on the relative performance of four coscheduling techniques and native local scheduling using a Linux cluster. The allocation factors that may affect the performance include Multi Programming Level (MPL), communication intensity and job placement. We have developed a generic, scalable and re-usable framework for implementing the coscheduling techniques on a cluster platform. Three prior coscheduling algorithms, Spin-Block (SB)[1][2], Dynamic Coscheduling (DCS)[3] and Periodic Boost (PB)[2], and a newly proposed coscheduling algorithm, called Co-ordinated Coscheduling (CC)[8] are implemented using this framework on a Myrinet connected 16-node Linux cluster. We use four NAS parallel benchmarks to analyze the impact of job allocation strategies. Our experimental results reveal that the blocking based schemes CC and SB consistently show tolerance to most allocation metrics and provide the best performance for all workloads. The local scheduling, and the two spin based techniques (DCS and PB) are more vulnerable to various job allocation strategies. Further, node sharing due to various job placement techniques seems a viable option for the CC and SB algorithms as the application execution times are little affected by such sharing.

The rest of this paper is organized as follows. Section 2 describes the implementation of three prior coscheduling techniques along with our proposed coscheduling algorithm using a generic framework. The job allocation metrics and performance results are presented in Section 3 followed by the concluding remarks in the last section.

2 A Generic Framework for Implementing Coscheduling Algorithms

In this section, first a brief description of the four coscheduling algorithms is given followed by our generic framework for their implementation. All coscheduling algorithms rely primarily on one of two local events (*arrival of a message* and *waiting for a message*) to determine when and which process to schedule. For example, in the SB algorithm, 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 [1][2]. Dynamic coscheduling algorithm (DCS) uses incoming messages to schedule the process for which the messages are destined [3]. 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, 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 [2].

Our new Co-ordinated Coscheduling (CC) scheme [8] is different in that it optimizes both sender and receiver side spinning to improve performance. With this scheme, the sender spins for a pre-determined amount of time waiting for an acknowledge from Network Interface Card (NIC). The NIC sends the acknowledge after pushing the message to the wire. If a send is completed within the spin time, the sender remains scheduled hoping that its receiver will be coscheduled and a response can be received soon. If the sender does not receive the acknowledge from the NIC within the spin time, it is blocked and the scheduler chooses another process from its runqueue. As soon as the NIC completes the corresponding send, it wakes up the original sender process, and makes it ready to be coscheduled before the reply comes from the other end. 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. The NIC firmware maintains per process message arrival information in a table and continuously updates the table by recording the cumulative number of incoming messages for the corresponding process. Every 10ms, the table information in the NIC is retrieved to find the process, which has the largest number of un-consumed incoming message, and the process is returned to the local scheduler to be run next. The *send_spin_time* and *recv_spin_time* are carefully calculated for maximal performance benefits.

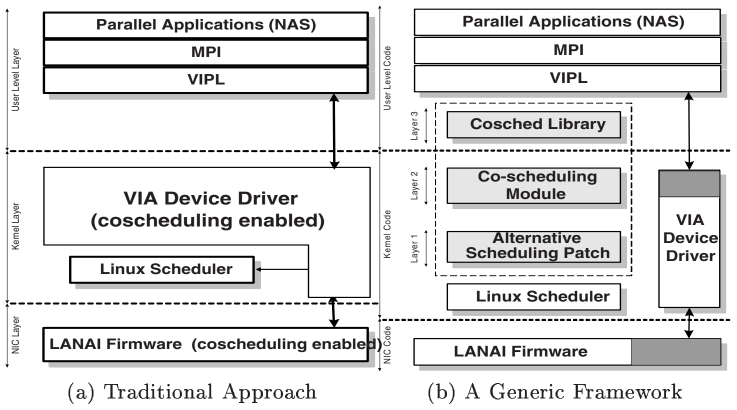


Fig. 1. Two Coscheduling Implementation Alternatives

Implementation of these schemes on a cluster needs significant modifications in the user-level communication layer such as VIA, NIC firmware and the device driver as shown in Figure 1 (a). Changing from one platform to another needs rework of the entire effort for each algorithm. In view of this, we present a generic, reusable framework that can be used to implement any coscheduling algorithm across various platforms, using well defined interfaces for the NIC, kernel and user layers [8]. This framework has been implemented on a Myrinet connected 16-node Linux cluster that uses industry standard VIA [6] as the user-level communication abstraction. In our framework (see Figure 1 (b)), an *alternative scheduling patch*, integrated in the local Linux scheduler, invokes the *coscheduling module* in which all the coscheduling algorithms used in this paper are implemented. The *coscheduling module* chooses the next possible running process based on the underlying coscheduling algorithm and provides appropriate interfaces to the VIA device driver and NIC firmware. The selected process is returned to the Linux native scheduler and the native scheduler intelligently decides whether to execute or ignore the new process, based on the overall system load. Otherwise, the Linux native scheduler selects on other process from its run-queue. Further details of CC scheme and the framework are also described in [8].

Table 1. Workload mixes used in this study.

Category	**Workload**	**Applications**	**Communication Intensity**
Parallel, Homogeneous	$Wl1$	(1,2,4,6)EPs	low
	$Wl2$	(1,2,4,6)LUs	Medium
	$Wl3$	(1,2,4,6)MGs	High
	$Wl4$	(1,2,4,6)CGs	Very high
Parallel, Heterogeneous	$Wl5$	EPs + MGs	Low + High
	$Wl6$	LUs + CGs	Medium + Very High

3 Job Allocation Strategies

This section explores three strategies that affect the way parallel jobs are allocated on a cluster. Their relative impacts are explained in detail through a series of experiments in subsequent sub-sections[1]. Before discussing the strategies, we explain the experimental testbed and the workload used in this study.

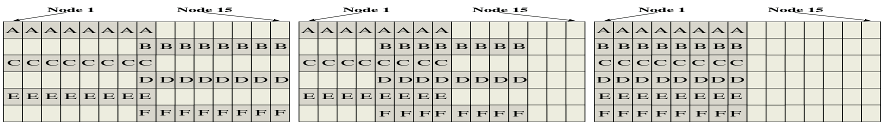
3.1 Experimental Platform and Workload

Our experimental testbed is a 16-node Linux (2.4.7-10) cluster, connected through a 16-port Myrinet [9] switch. Each node is an Athlon 1.76 GHZ uniprocessor machine, with 1 GB memory and a PCI based on-board intelligent NIC [9], with 8 MB of on-chip RAM and a 133 MHZ Lanai 9.2 RISC processor. We have significantly enhanced and used the Berkeley's VIA implementation (version 3.0) over Myrinet as our user-level communication layer and NERSC's MVICH (MPI-over-VIA) implementation [10] as our parallel programming library.

For our parallel workload, we consider 4 applications from the NAS parallel benchmark suite [11] : EP, LU, MG and CG; with lowest to highest communication intensities, respectively. Using combinations of these 4 applications, we designed a set of 6 parallel workloads as shown in Table 1. $Wl1$ through $Wl4$ exhibit uniform, homogeneous characteristics with low, medium, high and very high

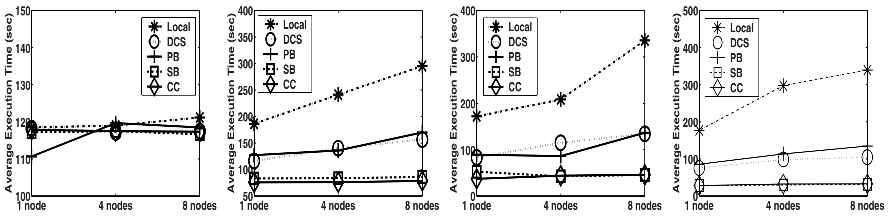
[1] In addition, we have analyzed the impact of CPU and I/O intensive several jobs on the performance of parallel jobs. The results are omitted here due to space limitation, but can be found in [8]

communication intensities respectively. $Wl5$ and $Wl6$ exhibit non-uniform, heterogeneous characteristics with mixed communication intensities. The numbers in the bracket represent the multi programming level of different applications. Total size of all applications, when run together, fits well within the memory (1GB), and hence, we incur no swapping overheads. Now, we examine various metrics that potentially influence the job-allocation decision using 4 coscheduling algorithms (DCS, PB, SB and CC).



(a) *Unbalanced* (b) *Partially Balanced* (c) *Fully Balanced*

Fig. 2. Sharing Strategies for allocating multiple jobs: (a) allocates to use all 15 nodes, overloading just 1 node, (b) overloads 4 nodes and frees up 3 nodes and (c) overloads 8 nodes to free up 7 extra nodes.



(a) $Wl1$ (b) $Wl2$ (c) $Wl3$ (d) $Wl4$

Fig. 3. Performance Analysis of sharing the number of nodes.

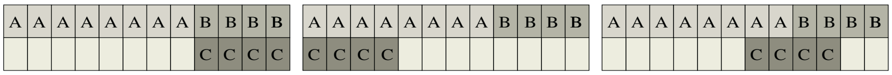
3.2 Effect of Job Placement

Typically, prior implementations of coscheduling algorithms have assumed the worst case scenario, where all parallel applications require maximum number of available nodes (16, in our case). Let us consider a more realistic case, where jobs have differing node requirements and not enough nodes are available to allocate all jobs independently. In such cases, we would like to see how sharing of multiple jobs can impact the overall performance. There are two basic issues to consider: (1) How many nodes are overloaded and (2) How many jobs are affected due to arrival of a new job?

For the first issue, considering the best, average and worst case of node sharing, we examine three job allocation techniques as shown in Figures 2 (a) through (c), respectively. We consider 6 class-A MG applications (each requires 8 nodes), and allocate them using three patterns (on available 15 nodes) as shown in Figure 2. Analyzing the results of these schemes under various workloads as shown in

Figures 3 (a) through (d), we find that only in the presence of a good coscheduling mechanism like CC [8] or SB [1][2], there is no difference on the average execution time of the 6 applications. Note that $Wl1$ is a very low communication workload, hence all coscheduling schemes perform almost the same. We expect that with a good coscheduling algorithm, we can choose any of the three allocation schemes and still achieve the best results. This conclusion is important because it can help optimize upon total number of nodes to be shared, and hence, can directly increase the overall throughput.

For the second issue, we consider three important cases: (1) Incoming job affects only 1 job of its own size (best case) (2) Incoming job affects 1 job, but of larger size (Intermediate case) and (3) Incoming job affects multiple (atleast two) jobs (Worst case). The three allocations are shown in Figures 4 (a) through (c), respectively, when A and B are resident jobs and C is the incoming job. Results of these allocation techniques are quite as expected : Figure 4 (a) is the best strategy in all cases, irrespective of any coscheduling algorithm used, followed by (b) and (c) respectively. We do not show these results for space limitations.

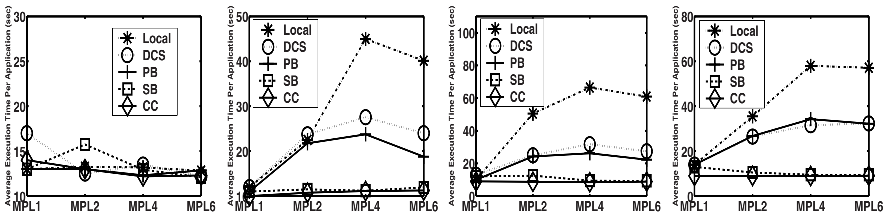


(a) *Share 1 job, same size* (b) *Share 1 job, larger size* (c) *Share Multiple jobs*

Fig. 4. Allocating a new job on nodes of a fully loaded machine

3.3 Effect of Multi-programming Level (MPL)

When coscheduling multiple jobs on cluster nodes, it is important to dynamically determine the maximum threshold number of parallel processes (thN) that can be successfully multi-tasked at a time. By successfully, we mean that the time taken by the thN processes when executed together should be less than or equal to the sum of the times they took, when executed in isolation. This metric is important from the allocation view-point because, if the allocator is aware of such a threshold, it can choose the nodes intelligently and avoid overloading on a node. Depending on the workload type of coscheduling scheme we use, such a threshold can vary. This is evidently seen in Figures 5 (a) through (d), where we plot the average execution time *per-application* of workloads $Wl1$ through $Wl4$, respectively, as we increase the MPL. We see that blocking based schemes like CC [8] and SB [1][2] are most tolerant to increase in the MPL, as the execution time *per-application* remains nearly constant. On the other hand, spin-based schemes suffer (sometimes drastically), especially in communication intensive workloads like $Wl2$, $Wl3$ and $Wl4$. This can be explained by the fact that as we increase the MPL, the likelihood of processes remaining coscheduled for a longer time gets lesser. This makes blocking and wakeup on demand a better option than spinning. We conclude from this result that in the presence of a good coscheduling algorithm (like CC or SB), we can allocate reasonably higher number of jobs (tested upto MPL 6) without performance penalty. Such schemes not only reduce the response time per-job, but also increase the overall system throughput, because multi-programming frees up other nodes.

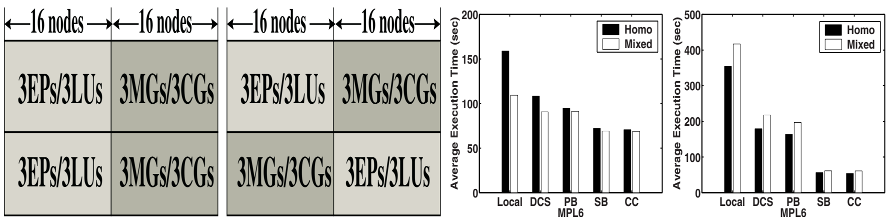


(a) $Wl1$ (b) $Wl2$ (c) $Wl3$ (d) $Wl4$

Fig. 5. Effect of MPL on the average execution time per application.

3.4 Effect of Variance in Communication Intensity

Another factor often neglected during job allocation is the communication intensity of the jobs. To see how it affects job allocation, we consider 2 types of workloads ($Wl5$ and $Wl6$), each mixed with jobs of varying communication intensities, and allocate them in two different ways: homogeneous and mixed, as shown in Figures 6 (a) and (b), respectively. Figures 6 (c) and (d) show the impact of various coscheduling algorithms on these job allocation strategies for workloads $Wl5$ and $Wl6$, respectively. We observe that for $Wl5$, mixed allocation is better, while for $Wl6$, homogeneous allocation is better. However, striking observation from both these results is that in the presence of a good coscheduling algorithm like CC and SB, both allocation schemes perform nearly the same. This leads us to the conclusion that with CC or SB, job allocator design gets simpler, as communication intensities have little impact on average job execution time.



(a) Homogeneous (b) Mixed (c) $Wl5$ (d) $Wl6$

Fig. 6. Allocation strategies [(a),(b)] and their effect on various job mixes [(c),(d)].

4 Conclusion and Future Work

Although several communication-driven coscheduling algorithms have been proposed for improving the performance of parallel jobs in a cluster environment, the allocation techniques that impact the relative merits of these algorithms have barely been studied. In this experimental work on a Myrinet connected 16-node Linux cluster and real workloads, we have analyzed several critical factors that can influence job-allocation decisions, and hence affect the overall performance. These factors include MPL, job placement strategies and communication intensity of parallel jobs. Several important conclusions about coscheduling algorithms can be derived from our experimental results. First, when not enough nodes are available to execute jobs independently, node-sharing becomes critical for performance. We find that in the presence of coscheduling mechanisms like CC and SB, maximizing node-sharing is a good option as the per-application execution time remains nearly constant. However, with other mechanisms (Local, DCS, PB) the execution time increases with sharing, hurting the overall performance. Moreover, sharing should be done intelligently not to affect more than a single parallel job. The node sharing was tested upto a relatively high MPL of six, and with CC and SB, there was no adverse impact on the performance of any parallel job. Second, we find that mixing jobs of differing communication intensities does not hurt performance in the presence of blocking-based coscheduling algorithms like CC and SB. This strengthens the case for using blocking-based coscheduling algorithms even further.

References

1. A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring, "Scheduling with Implicit Information in Distributed Systems," in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1998.
2. S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das, "A Closer Look at Coscheduling Approaches for a Network of Workstations," in *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 96–105, June 1999.
3. 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*, pp. 231–256, March 1998. LNCS 1459.
4. T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.
5. S. Pakin, V. Karamcheti, and A. A. Chien, "Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs.," *IEEE Concurrency*, vol. 5, pp. 60–72, April-June 1997.
6. Compaq, Intel and Microsoft Corporations, "Virtual Interface Architecture Specification. Version 1.0," Dec 1997. Available from `http://www.vidf.org`.
7. Dror G. Feitelson, Larry Rudolph, "Mapping and scheduling in a shared parallel environment using distributed hierarchical control," in *International Conference on Parallel Processing*, vol. 1, pp. 1–8, 1990.

8. S. Agarwal, "A Generic Infrastructure for Coscheduling Mechanisms on Clusters," Dec 2002. M.S. Thesis. Available from `http://www.cse.psu.edu/ sagarwal/sagarwalMSThesis.pdf`.
9. N. J. Boden *et al.*, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro*, vol. 15, pp. 29–36, February 1995.
10. National Energy Research Scientific Computing Center, "M-VIA: A High Performance Modular VIA for Linux," 2001. Available from `http://www.nersc.gov/research/FTG/via/`.
11. N. A. S. division., "The nas parallel benchmarks (tech report and source code)." Available from `http://http://www.nas.nasa.gov/Software/NPB/`.