Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000).

Scheduling with Global Information in Distributed Systems [*]

Fabrizio Petrini[†] and Wu-chun Feng[†§]
{fabrizio, feng}@lanl.gov

[†] Computing, Information, and Communications Division
Los Alamos National Laboratory
Los Alamos, NM 87545

[§] School of Electrical & Computer Engineering
Purdue University
W. Lafayette, IN 47907

Abstract

Buffered coscheduling is a distributed scheduling methodology for time-sharing communicating processes in a distributed system, e.g., PC cluster. The principle mechanisms involved in this methodology are communication buffering and strobing. With communication buffering, communication generated by each processor is buffered and performed at the end of regular intervals (or time slices) to amortize communication and scheduling overhead. This regular communication structure is then leveraged by introducing a strobing mechanism which performs a total exchange of information at the end of each time slice. Thus, a distributed system can rely on this global information to more efficiently schedule communicating processes rather than rely on isolated or implicit information gathered from local events between processors.

In this paper, we describe how buffered coscheduling is implemented in the context of our SMART simulator. We then present performance measurements for two synthetic workloads and demonstrate the effectiveness of buffered coscheduling under different computational granularities, context-switch times, and time-slice granularities.

Keywords*: distributed resource management, parallel job scheduling, distributed operating systems, co-scheduling, gang scheduling.*

[*]This work was supported by the U.S. Dept. of Energy through Los Alamos National Laboratory contract W-7405-ENG-36. This paper is LA-UR 00-894.

1. Introduction

The scheduling of parallel jobs has long been an active area of research [7, 8]. It is a challenging problem because the performance and applicability of parallel scheduling algorithms is highly dependent upon factors at different levels: the workload, the parallel programming language, the operating system (OS), and the machine architecture.

Time-sharing scheduling algorithms are particularly attractive because they can provide good response time without migration or predictions on the execution time of the parallel jobs. However, to achieve good performance, time-sharing algorithms require communicating processes to be scheduled simultaneously. This is a critical problem because the software communication overhead and the scheduling overhead to wake up a sleeping process dominate the communication time on most parallel machines.

In recent years, researchers have developed parallel scheduling algorithms that can be loosely organized into three main classes, according to the degree of coordination between processors: *explicit coscheduling, local scheduling* and *implicit or dynamic coscheduling*.

On the one end of the spectrum, explicit coscheduling [6] ensures that the scheduling of communicating jobs is coordinated by creating a static global list of the order in which jobs should be scheduled and then requiring a simultaneous context-switch across all processors. Unfortunately, this approach is neither scalable nor reliable. Furthermore, it requires that the schedule of communicating processes be precomputed, thus complicating the coscheduling of applications and requiring pessimistic assumptions about which processes communicate with one another. Lastly, explicit coscheduling of parallel jobs also adversely affects performance on interactive and I/O-based jobs [13].

At the other end of the spectrum is local scheduling, where each processor independently schedules its processes. While this approach is attractive due to its ease of construction, the performance of fine-grain communicating jobs is severely impacted because scheduling is not coordinated across processors [10].

An intermediate approach developed at UC Berkeley and MIT in recent years is implicit or dynamic coscheduling [1, 5, 15, 20]. With implicit coscheduling, each local scheduler makes independent decisions that dynamically coordinate the scheduling actions of cooperating processes across processors. These actions are based on local events that occur naturally within communicating applications. For example, on message arrival, a processor speculatively assumes that the sender is active and will likely send more messages in the near future. The implicit information available for implicit coscheduling consists of two inherent events: response time and message arrival [1].

The programming model used in the implementation of implicit coscheduling does not support a full-fledged communication library as MPI and considers only three basic communication operations: *reads* and *writes*, request-response messages between pairs of processes requiring the requesting process to wait for the response, and *barriers* to synchronize all processes.

The limitations of the above localized flow-control strategy emerge when processes perform continuous reads or writes in an irregular communication pattern, e.g., they can flood the output buffers with write operations [1]. Some of these limitations are addressed in [14] with a technique called *periodic boost*. Instead of sending an interrupt for each incoming message, the kernel periodically examines the status of the network interface, thus reducing the overhead with high communication workloads. Our methodology is based on a similar buffering technique which is integrated with global time-slicing and and a strobing algorithm.

The rest of the paper is organized as follows. Section 2 provides the motivation for our buffered coscheduling methodology. The methodology itself is described in Section 3 and some preliminary results are presented in Section 4. Finally, we present our conclusions in Section 5.

2. Motivation

Figure 1 shows the global processor and network utilization (i.e., the number of active processors and the fraction of active links) during the execution of a transpose FFT algorithm on a parallel machine with 256 processors. These processors are connected with an indirect interconnection network using state-of-the-art routers [3]. Based on these figures, there is obviously an *uneven and inefficient use of system resources*. During the two computational phases of the transpose, the network is idle. Conversely, when the network is actively transmitting messages, the processors are essentially idle. These characteristics are shared by many SPMD programs, including Accelerated Strategic Computing Initiative (ASCI) application codes such as Sweep3D [11]. Hence, there is tremendous potential for increasing resource utilization in a parallel machine.

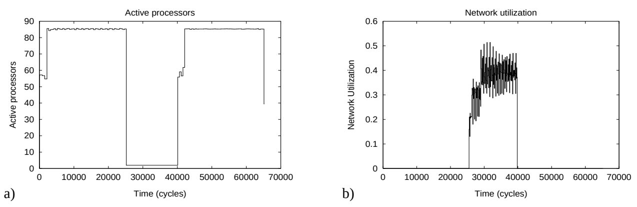

Figure 1. Resource Utilization in a Transpose FFT Algorithm.

Another important characteristic shared by many parallel programs is their access pattern to the network. The vast majority of parallel applications display *bursty communication patterns* with alternating spikes of impulsive communication with periods of inactivity [16]. Thus, there exists a significant amount of unused network bandwidth which could be used for other purposes.

3 Multitasking Parallel Jobs

In order to improve the resource utilization of parallel programs, we propose our buffered coscheduling methodology to multitask parallel jobs and allow all the communication and I/O which arise from a *set* of parallel programs to be overlapped with the computations in those programs. Buffered coscheduling consists of three techniques: communication buffering, strobing, and non-blocking, one-sided communication.

3.1 Communication Buffering

Rather than incurring communication and scheduling overhead on a per-message basis, we propose to accumulate the communication messages generated by each processor and amortize the overhead over a set of messages. Specifically, the cost of the system calls necessary to access the kernel data structures for communication is amortized over a set of system calls rather than being incurred on each individual system call. This implies that our methodology can be tolerant to the potentially high latencies that can be introduced in a kernel call or in the initialization of the NIC that can reside on a slow I/O bus. Also, by delaying the communication, we allow for the global scheduling of the communication pattern. And because we can implement zero-copy (or low-copy, if we desire fault-tolerant communication) communication, our approach to communication buffering can theoretically achieve performance comparable to user-level network interfaces (i.e., OS-bypass protocols) [2] without using specialized hardware.

3.2 Strobing

The uneven resource utilization and the periodic, bursty communication patterns generated by many parallel applications can be exploited to perform a total exchange of information and a synchronization of processors at regular intervals with little additional cost. This provides the parallel machine with the capability of filling in communication holes generated by parallel applications.

In order to provide the above capability, we propose a strobing mechanism to support the scheduling of a set of parallel jobs which share a parallel machine. Let us assume that each parallel job runs on the entire set of p processors, i.e., jobs are time-sharing the whole machine. At a high level, the strobing mechanism performs an optimized total-exchange of control information which then triggers the downloading of any buffered packets into the network.

The strobe can be implemented by designating one of the processors as the *master*, the one who generates the "heartbeat" of the strobe. The generation of heartbeats is achieved by using a timeout mechanism which can be associated with the network interface card (NIC). This ensures that strobing incurs little CPU overhead as most NICs can count down and send packets asynchronously. This is true for a wide range of NICs, ranging from simple 100-Mb/s Ethernet cards to more sophisticated cards such as Myrinet [3].

On reception of the heartbeat, each processor (excluding the master) is interrupted and downloads a broadcast heartbeat into network. After downloading the heartbeat, the processor continues running the currently active job. (This ensures computation is overlapped with communication.) When p heartbeats arrive at a processor, the processor enters a strobing phase where its kernel downloads any buffered packets to the network[1].

Figure 2 outlines how computation and communication can be scheduled over a generic processor. At the beginning of the heartbeat, t_0, the kernel downloads control packets for the total exchange of information. During the execution of the barrier synchronization, the user process then regains control of the processor; and at the end of it, the kernel schedules the pending communication accumulated before

[1]Each heartbeat contains information on which processes have packets ready for download and which processes are asleep waiting to upload a packet from a particular processor. This information is characterized on a per-process basis so that on reception of the heartbeat, every processor will know which processes have data heading for them and which processes on that processor they are from.

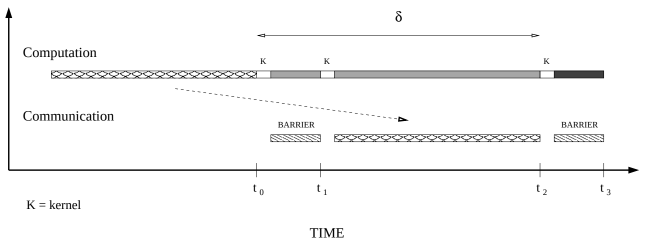

Figure 2. Scheduling Computation and Communication. The communication accumulated before t_0 is downloaded into the network between t_1 and t_2.

t_0 to be delivered in the current time slice, i.e., δ. At t_1, the processor will know the number of incoming packets that it is going to receive in the communication time-slice as well as the sources of the packets and will start the downloading of outgoing packets.

This strategy can be easily extended to deal with space-sharing where different regions run different sets of programs [6, 12, 21]. In this case, all regions are synchronized by the same heartbeat.

The total exchange of information can be properly optimized by exploiting the low-level features of the interconnection network. For example, if control packets are given higher priority than background traffic at the sending and receiving endpoints, they can be delivered with predictable network latency[2] during the execution of a direct total-exchange algorithm[3] (Figure 3). We generated this distribution using a network of 256 processing nodes equipped with wormhole routers similar to those in the SGI Origin 2000 and assumed the existence of random background traffic that occupies 80% of the network capacity. If control packets are prioritized at the network endpoints, they can be delivered with a bounded latency of 30 μs.

We also analyzed the execution time of the direct total-exchange algorithm in a family of indirect networks with up to 1024 processing nodes. In this experiment, whose results are shown in Figure 4, we assume the existence of background traffic that varies from 20% to 80% of the network capacity. We can see that the execution time is largely insensitive to the intensity of the background traffic. With 64 processing nodes (the configuration of a single SGI Origin 2000 cluster) the execution time is only 50 μs, and this increases to 150 μs with 256 nodes. Due to the quadratic increase in the number of messages sent during the total exchange, the execution time reaches 1 ms with 1024 nodes,

[2]The network latency is the time spent in the network without including source and destination queueing delays.

[3]In a direct total-exchange algorithm, each packet is sent directly from source to destination, without intermediate buffering.

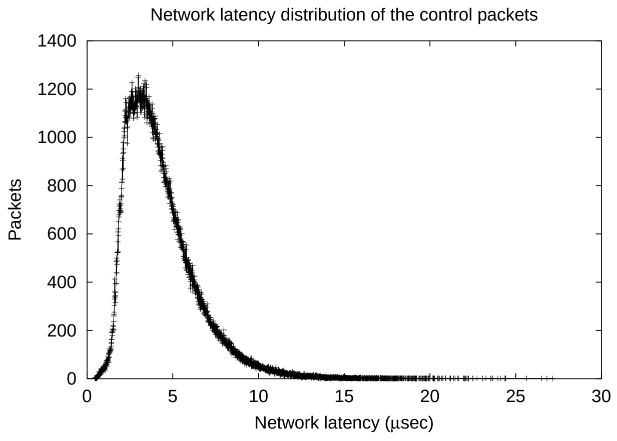

Figure 3. Network Latency Distribution.

limiting the scalability of the approach.

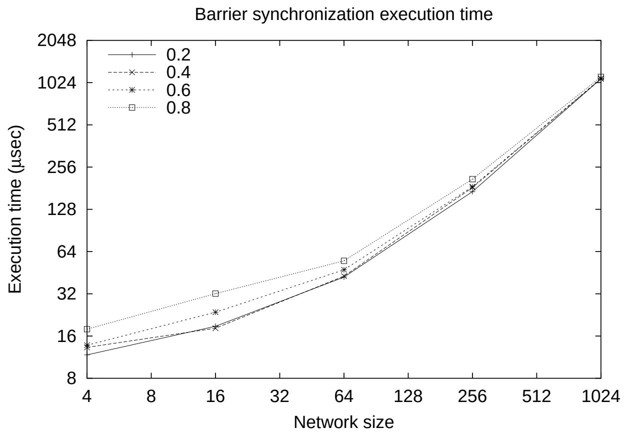

Figure 4. Execution time of the total exchange algorithm in a family of interconnection networks with up to 1024 processing nodes.

This scalability problem can be addressed in a clustered architecture like ASCI Blue Mountain by using a multi-phase, indirect algorithm. In the first phase, we perform a total exchange *within* each cluster. Next, we do a total exchange *between* clusters. Finally, we conclude with a final phase internal to the clusters, giving a barrier synchronization time of less than 300 μs.

The global knowledge of the communication pattern provided by the total exchange allows for the implementation of efficient flow-control strategies. For example, it is possible to avoid congestion inside the network by carefully scheduling the communication pattern and limiting the negative effects of hot spots by damping the maximum amount of information addressed to each processor during a time-slice. The same information can be used at the kernel level to provide fault-tolerant communication. For example, the knowledge of the number of incoming packets greatly simplifies the implementation of receiver-initiated recovery protocols.

3.3 Blocking vs. Non-Blocking

One of the most limiting constraints in the implementation of time-sharing algorithms is the need to schedule simultaneously communicating processes. This problem is exacerbated with blocking communication, which imposes an explicit handshake between sender and receiver.

We argue that this problem can eliminated, or at least alleviated, by slightly modifying the communication structure of parallel jobs and replacing blocking communication with non-blocking primitives and/or one-sided communication.

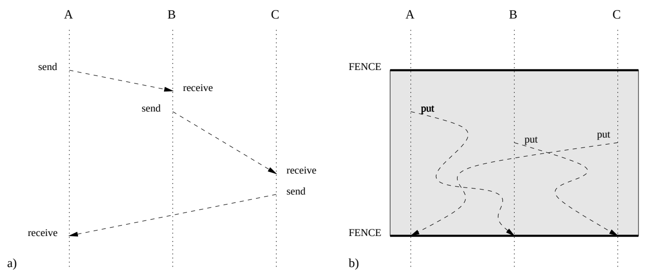

Figure 5. (a) Message Passing. (b) One-Sided Communication.

Let us consider the following example. The dynamics of a message-passing program can be represented as a two-dimensional graph with processes on the horizontal axis and time on the vertical one, as shown in Figure 5. Arrows between processes represent communication between a sender and a receiver. In Figure 5(a), three processes exchange messages. For the sake of convenience, let us assume that there is no dependency between the messages (i.e., they can be sent in any order). Using a traditional, blocking, message-passing programming style, we must define a communication schedule even if one is *not* required, e.g., A sends to B, B receives from A and sends to C, C receives from B and sends to A.

With one-sided communication (or non-blocking communication primitives, in general), the actual message transmission and the synchronization are decoupled, leaving many degrees of freedom to re-arrange message transmission. In Figure 5(b), the same communication pattern is delimited by two *barriers* which include the communication executed with *put* primitives. The communication can be executed in any order, provided that the information is delivered at the end of the synchronization calls. Also, communicating processes do *not* need to be simultaneously scheduled to perform the communication.

3.4 Bulk-Synchronous Parallel Programs

Using our proposed strobing and buffering mechanisms, any generic parallel program can be transformed into a Bulk-Synchronous Parallel (BSP) one [19]. Although the buffering and strobing mechanisms alone improve parallel program performance, transforming a parallel program into a BSP one not only can improve performance further but also allows for accurate prediction of the execution times.

A BSP computation consists of a sequence of parallel *supersteps*. During a superstep, each processor can perform a number of computation steps on values held locally at the beginning of the superstep and can issue various remote read and write requests that are buffered and delivered at the end of the superstep. This implies that communication is clearly separated from synchronization, i.e. it can be performed in any order, provided that the information is delivered at the beginning of the following superstep. However, while the supersteps in the original BSP model can be variable in length, our programming model generates computation and communication slots which are fixed in length and are determined by the time-slice.

One important benefit of the BSP model is the ability to accurately predict the execution time requirements of parallel algorithms and programs. This is achieved by constructing analytical formulae that are parameterized by a few constants which capture the computation, communication, and synchronization performance of a p-processor system. These results are based on the experimental evidence that the generic collective communication pattern generated by a superstep called h-relation[4] can be routed with predictable time [9, 17]. This implies that the maximum amount of information sent or received by each processor during a communication time-slice can be statically determined and enforced at run time by a global communication scheduling algorithm. For example, if the duration of the time-slice is δ and the permeability of the network (i.e., the inverse of the aggregate network bandwidth) is g, the upper bound h_{max} of information, expressed in bytes, that can be sent or received by a single processor is $h_{max} = \frac{\delta}{g}$. Furthermore, by globally scheduling a communication pattern, as described in Section 3.2, we can derive an accurate estimate of the communication time with simple analytical models already developed for the BSP model [4].

[4] h denotes the maximum amount of information sent or received by any process during the superstep.

Another important benefit of the BSP model is higher resource utilization over the parallel machine, irrespective of the computational and communication patterns. For example, a sparse communication pattern (where a single processor receives h_{max} bytes) or a more dense communication pattern (where more processors share the same upper bound) can be routed in the same communication time-slice. This means that it is possible to use spare communication bandwidth to deliver packets generated by other parallel jobs without detrimental effects. More generally, as with any multiprogrammed system, multitasking a collection of bad (parallel) programs, i.e., unbalanced computation or communication, may produce the same behavior as a single well-behaved (parallel) program. Multitasking can provide opportunities for filling in “spare communication cycles” by merging sparse communication patterns together to produce a denser communication pattern.

Lastly, the BSP model is also beneficial for fault tolerance[5]. Fault tolerance can be naturally implemented by checkpointing the machine at the synchronization points at the end of a time-slice.

4 Experimental Results

Our preliminary results include a working implementation of a representative subset of MPI-2 on a detailed (register-level) simulation model [18]. The simulation environment includes a standard version of MPI-2 and a multitasking one that implements the main features of our proposed methodology.

4.1 Characteristics of the Synthetic Workloads

As in [5], the workloads used consist of a collection of single-program multiple-data (SPMD) parallel jobs that alternate phases of purely local computation with phases of interprocess communication. A parallel job consists of a group of P processes where each process is mapped onto a processor throughout its execution. Processes compute locally for a time uniformly selected in the interval $(g - \frac{v}{2}, g + \frac{v}{2})$. By adjusting g, we model parallel programs with different computational granularities; and by varying v, we change the degree of load-imbalance across processors. The communication phase consists of an opening barrier, followed by an optional sequence of pairwise communication events separated by small amounts of local computation, c, and finally an optional closing barrier.

We consider two communication patterns: *Barrier* and *Transpose*. *Barrier* consists of only the opening barrier and thus contains no additional dependencies. This workload

[5] This is of vital importance to the large ASCI supercomputers where the MTBF can be on the order of hours.

can be used to analyze how our methodology responds to load imbalance. *Transpose* is a communication-intensive workload that emulates the communication pattern generated by the FFT transpose algorithm

For our synthetic workload, we consider three parallel jobs with the same computational granularity, load imbalance, and communication pattern arriving at the same time in the system. The communication granularity, c, is fixed at 8 μs. The number of communication/computation iterations is scaled so that each job runs for approximately 1 second in a dedicated environment. The system consists of 32 processors, and each job requires 32 processes (i.e. jobs are only time-shared).

4.2 The Simulation Model

The simulation tool that we used in our experimental evaluation is called SMART (Simulator of Massive ARchitectures and Topologies) [18], a flexible tool designed to model the fundamental characteristics of a massively parallel architecture.

The current version of SMART is based on the x86 instruction set. The architectural design of the processing nodes is inspired by the Pentium II family of processors. In particular, it models a two-level cache hierarchy with a write-back L1 policy and non-blocking caches.

For our experiments, we assume a network of 32 processors, each running at 500 MHz, interconnected in a 5-dimensional cube topology with performance characteristics similar to those of Myrinet routing and network cards [3]. This network features a one-way data rate of about 1 Gb/s and a base network latency of few μs.

The run-time support running on this simulated platform includes a standard version of a substantive subset of MPI-2 and a multitasking version of the same subset that performs the strobing algorithm at the end of each time-slice as outlined in Section 3. It is worth noting that the multitasking MPI-2 version is actually much simpler than the sequential one because the buffering of the communication primitives greatly simplifies the run-time support.

4.3 Sensitivity Analysis

Figures 6 and 7 illustrate the communication and computation characteristics of our synthetic benchmarks as a function of the communication pattern, granularity, load-imbalance, time-slice duration, and context-switch penalty. Each bar shows the percentage of time spent in one of the following states, averaged over all processors: computing, context-switching and idling.

For each communication pattern, we analyze the Cartesian product of nine alternatives generated by considering time-slices of 500 μs, 1 ms and 2 ms with context-switch penalties of 50, 100, and 200 μs. For each alternative, we reduce the computational grain size g, going from left to right, from 50 ms down to 100 μs and consider ”six groups of three bars” of experiments. Each group has the same computational granularity, and the load imbalance is increased as a function of the granularity itself. We consider three cases: $v = 0$ (i.e. no variance), $v = g$ (in this case the variance is equal to the computational granularity) and $v = 2g$ (high degree of imbalance).

At the bottom of each figure we also report the breakdown for the same communication pattern when the workload is run in dedicated mode with standard MPI-2 run-time support (i.e., a single job is run until completion without multitasking). A black square under a bar highlights the configurations where the multitasking approach produces better resource utilization than the standard approach.

Based on Figures 6 and 7, we make the following observations. First, the performance of buffered coscheduling is sensitive to the context-switch latency. As context-switch latency decreases, resource utilization and throughput improve. Second, as the load imbalance of a program increases, the idle time increases. Third, and most importantly, these initial results indicate that the time-slice length is a critical parameter in determining overall performance. A short time-slice can achieve excellent load balancing even in the presence of highly unbalanced jobs. The downside is that it amplifies the context-switch latency. On the other hand, a long time-slice can virtually hide all the context-switch latency, but it cannot reduce the load imbalance, particularly in the presence of fine-grained computation.

More specifically, Figure 6(g) shows that a relatively small time-slice coupled with a small context-switch latency generally results in better processor utilization than a single job running in a dedicated environment (Figure 6(l)) in eleven cases out of eighteen. Running a single job provides only slightly better (less than 10%) performance with perfect load balancing ($v = 0$) because we have to pay the context-switch penalty without improving the load balance. On the other hand, in the presence of load imbalance, job multitasking can smooth out the differences in the load.

As a rule of thumb, buffered coscheduling performs admirably as long as the average computational grain size is larger than the time-slice, and the time-slice in turn is sufficiently larger than the context-switch penalty. In addition, when the average computational grain size is larger than the time-slice, the processor utilization is mainly influenced by the degree of imbalance.

In these initial experimental results, we did not take into account the effects of the memory hierarchy on the working sets of different jobs. As a consequence, buffered coscheduling requires a larger main memory in order to avoid memory swapping. We consider this as the main limitation of our approach.

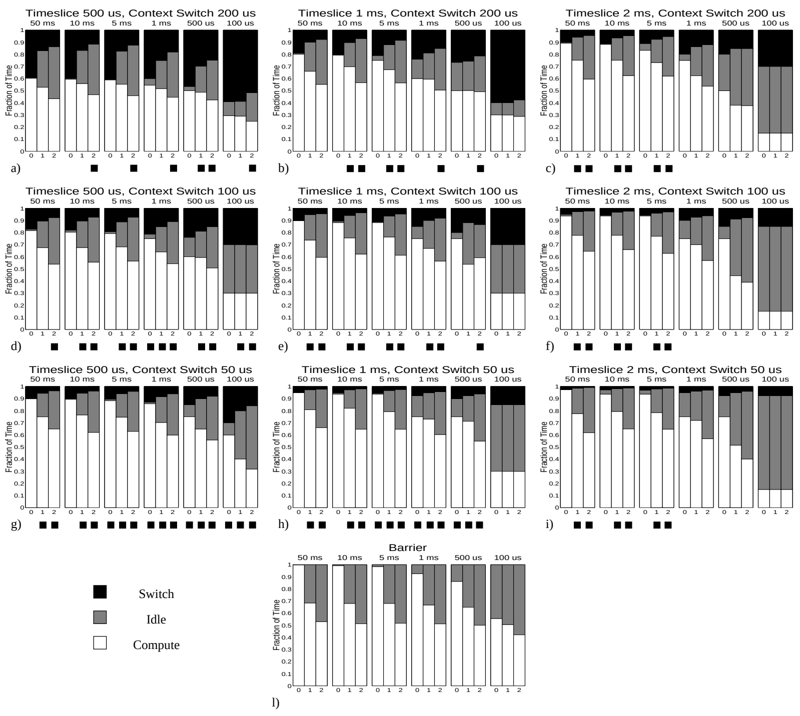

Figure 6. Execution Characteristics of the *Barrier* Workload.

5. Conclusion

In this paper we have presented buffered coscheduling, a new methodology to multitask parallel jobs on a parallel computer. Buffered coscheduling represents a significant improvement over existing work reported in the literature. It allows for the implementation of a global scheduling policy, as done in explicit coscheduling, while maintaining the overlapping of computation and communication provided by implicit coscheduling.

We initially addressed the complexity of a huge design space using two families of synthetic workloads. The preliminary experimental results reported in this paper show that our methodology can provide better resource utilization, particularly in the presence of load imbalance and communication-intensive jobs.

We plan to extend these preliminary results by considering the effects of the memory hierarchy in a real application rather than in synthetic workloads and to implement a multitasking version of MPI-2 in a Linux cluster.

References

[1] A. C. Arpaci-Dusseau, D. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[2] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawick, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995.

[4] D. C. Burger and D. A. Wood. Accuracy vs. Performance in Parallel Simulation of Interconnection Networks. In *Proceedings of the 9th International Parallel Processing Symposium, IPPS'95*, Santa Barbara, CA, April 1995.

[5] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, May 1996.

[6] D. G. Feitelson and M. A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[7] D. G. Feitelson and L. Rudolph. Parallel job scheduling: issues and approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[8] D. G. Feitelson and L. Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In D. G. Feitelson

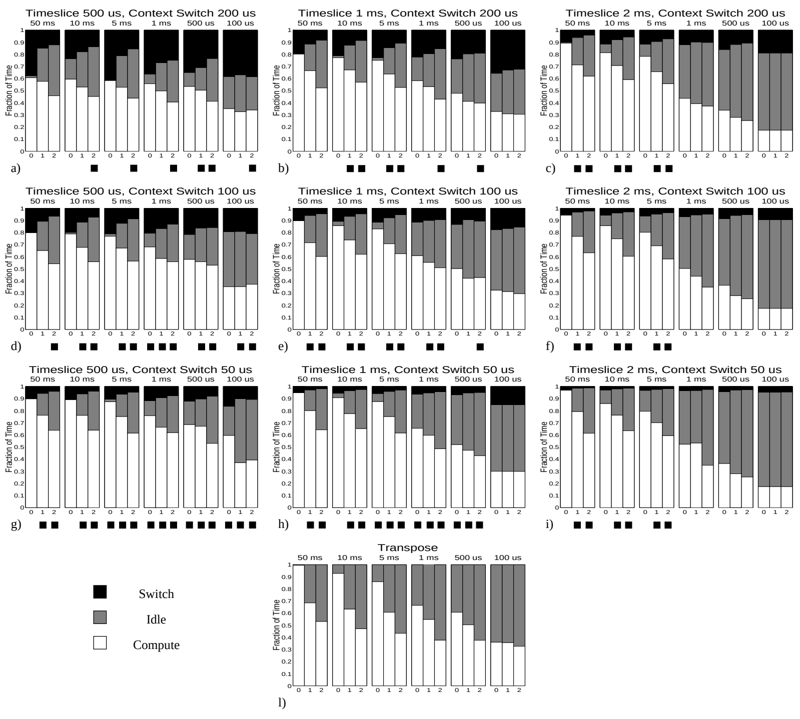

Figure 7. Execution Characteristics of the *Transpose* Workload.

and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[9] A. Gerbessiotis and F. Petrini. Network Performance Assessment under the BSP Model. In *International Workshop on Constructive Methods for Parallel Programming, CMPP'98*, Marstrand, Sweden, June 1998.

[10] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference*, pages 120–132, May 1991.

[11] A. Hoisie, O. Lubeck, and H. Wasserman. Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters. In *The Ninth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99)*, Annapolis, MD, February 1999.

[12] M. A. Jette. Performance Characteristics of Gang Scheduling in Multiprogrammed Environments. In *Supercomputing 97*, San Jose, CA, November 1997.

[13] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[14] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. A Closer Look At Coscheduling Approaches for a Network of Workstations. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures, SPAA'99*, Saint-Malo, France, June 1999.

[15] W. E. W. Patrick Sobalvarro, Scott Pakin and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 231–256. Springer-Verlag, 1998.

[16] F. Petrini. Network Performance with Distributed Memory Scientifi c Applications. Submitted to the Journal of Parallel and Distributed Computing, September 1998.

[17] F. Petrini and M. Vanneschi. Effi cient Personalized Communication on Wormhole Networks. In *The 1997 International Conference on Parallel Architectures and Compilation Techniques, PACT'97*, San Francisco, CA, November 1997.

[18] F. Petrini and M. Vanneschi. SMART: a Simulator of Massive ARchitectures and Topologies. In *International Conference on Parallel and Distributed Systems Euro-PDS'97*, Barcelona, Spain, June 1997.

[19] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Journal of Scientifi c Programming*, 1998.

[20] P. Sobalvarro and W. E. Weihl. Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the 9th International Parallel Processing Symposium, IPPS'95*, Santa Barbara, CA, April 1995.

[21] K. Suzaki and D. Walsh. Implementing the Combination of Time Sharing and Space Sharing on AP/Linux. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1998.