Journal of Parallel Algorithms & Applications, 2000. LA-UR 00-3476

Improved Resource Utilization with Buffered Coscheduling

Fabrizio Petrini[⋆] and Wu-chun Feng[⋆†]

{fabrizio, feng}@lanl.gov

[⋆] Computing, Information, & Communications Division
Los Alamos National Laboratory
Los Alamos, NM 87545

[†] School of Elec. & Comp. Engg.
Purdue University
W. Lafayette, IN 49707

Abstract

We present *buffered coscheduling*, a new methodology to multitask parallel jobs in a message-passing environment and to develop parallel programs that can pave the way to the efficient implementation of a distributed operating system. Buffered coscheduling is based on three innovative techniques: communication buffering, strobing, and non-blocking communication. By leveraging these techniques, we can perform effective optimizations based on the global status of the parallel machine rather than on the limited knowledge available locally to each processor.

The advantages of buffered coscheduling include higher resource utilization, reduced communication overhead, efficient implementation of flow-control strategies and fault-tolerant protocols, accurate performance modeling, and a simplified yet still expressive parallel programming model which offloads many resource-management tasks to the operating system. Preliminary experimental results show that buffered coscheduling is very effective in increasing the overall performance in the presence of load imbalance and communication-intensive workloads.

Keywords: Parallel Job Scheduling, Distributed Operating Systems, Communication Protocols.

1 Introduction

The scheduling of parallel jobs across a parallel or distributed system has long been an active area of research [8, 9]. It is a challenging problem because the performance and applicability of parallel scheduling algorithms is highly dependent upon factors at different levels: workload, parallel programming language, operating system (OS), and machine architecture.

Time-sharing scheduling algorithms are particularly attractive because they can provide good response time without migration or prediction on the execution time of the parallel jobs. However, time-sharing has the drawback that *communicating processes must be scheduled simultaneously to achieve good performance.* With respect to performance, this is a critical problem because the communication overhead and the scheduling overhead to wake up a sleeping process dominate the communication time on most parallel machines [15].

Over the 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 [7] ensures that the scheduling of communicating jobs is coordinated by constructing a static global list of the order in which jobs should be scheduled. A simultaneous context-switch is then required across all processors. Unfortunately, this straightforward methodology is neither scalable nor reliable. Furthermore, explicit coscheduling requires that the schedule of communicating processes be precomputed, which complicates the coscheduling of client-server applications and requires pessimistic assumptions about which processes communicate with one another. Finally, explicit coscheduling of parallel jobs interacts poorly with interactive jobs and jobs performing I/O [16].

At the other end of the spectrum is local scheduling, where each processor independently schedules its processes. While this methodology is attractive due to its ease of construction, the performance of fine-grained communication jobs can be orders of magnitude worse than with explicit coscheduling because the scheduling is not coordinated across processors [11].

In recent years, UC Berkeley and MIT have introduced an intermediate approach called implicit or dynamic coscheduling [1, 6, 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, the receiving processor speculatively assumes that the sender is active and will probably 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]. An in-depth performance analysis of coscheduling strategies can be found in [18].

Response time is the time for the response to a message request to return to the sending process. Assuming the destination process must be scheduled for a response to be returned, a fast response indicates to the sending node that the corresponding destination process is probably currently scheduled. Therefore, the desired action for implicit coscheduling is to keep the sender scheduled. Conversely, if the response is not received in a timely fashion, the sending node can infer that the destination is probably *not* scheduled. Thus, it is not beneficial to keep the sender scheduled.

The mechanism that achieves these desired actions is *two-phase spin blocking.* With two-phase spin-blocking, a process spins for some threshold amount of time, and if the response arrives before the time expires, it continues executing. If the response is not received within the threshold, the process voluntarily relinquishes the processor so a competing process can be scheduled.

The other inherent event used in implicit coscheduling is message arrival, the receipt of a message from a remote node. When a message arrives, the implication is that the corresponding remote process was recently scheduled. Therefore, it may be beneficial to schedule, or keep scheduled, the receiving process and to increase its spin time.

The main drawbacks of dynamic and implicit coscheduling include (1) the limited programming model supported, (2) the limitation of a localized flow-control strategy, (3) the non-trivial implementation of fault tolerance, and (4) the lack of a reliable performance model of the execution time of parallel jobs due to the dynamic interleaving of several jobs. Some of the above limitations are successfully addressed in [18] with a technique called *Periodic Boost*. Rather than sending an interrupt for each incoming message, the kernel periodically examines the status of the network interface, thus reducing the overhead for communication-intensive workloads.

In contrast, we present a new methodology which exploits the positive aspects of both explicit and implicit coscheduling using three innovative techniques: communication buffering, strobing, and non-blocking, one-sided communication. By leveraging these techniques, we can perform effective optimizations based on the status of the parallel machine rather than on the limited knowledge available locally to each processor.

The benefits of buffered coscheduling include higher resource utilization, dramatic simplification of the run-time support, reduced communication overhead, efficient global implementation of flow-control strategies and fault-tolerant protocols, accurate performance modeling, and a simplified yet still expressive parallel programming model.

The rest of the paper is organized as follows. Section 2 characterizes important properties which are shared by many parallel applications and which inspired our buffered coscheduling approach. Buffered coscheduling itself is described in Section 3, and some preliminary results are presented in Section 4. Finally, we present our conclusions in Section 5.

2 Resource Utilization of Parallel Programs

In Figure 1, we show the network utilization by running four distinct applications over a parallel machine with 256 processors [22]. In all four cases, we can identify *communication holes*, i.e., periods of network inactivity, in the network. The processors are connected with an indirect interconnection network using state-of-the-art routers. Based on these figures, there is obviously an *uneven and inefficient use of system resources*. These characteristics are shared by many SPMD programs, including Accelerated Strategic Computing Initiative (ASCI) application codes such as Sweep3D [13]. Hence, there is tremendous potential for increasing resource utilization in a parallel machine.

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 [22].

Thus, there exists a significant amount of communication bandwidth which can be made available for other purposes.

3 Buffered Coscheduling

To improve the resource utilization of parallel programs, we propose to multitask parallel jobs. That is, instead of overlapping computation with communication and I/O within a *single parallel program*, all the communication and I/O which arises from a *set of parallel programs* can be overlapped with the computations in those programs. To implement this multitasking, we use a buffered coscheduling approach which relies on three techniques. First, the communication generated by each processor is buffered and performed at the end of regular intervals (or time-slices) in order to amortize the communication and scheduling overhead. By delaying communication, we allow for the global scheduling of the communication pattern. Second, a strobing mechanism performs a total exchange of control information at the end of each time-slice so that massively parallel machines may move away from isolated scheduling algorithms [1]

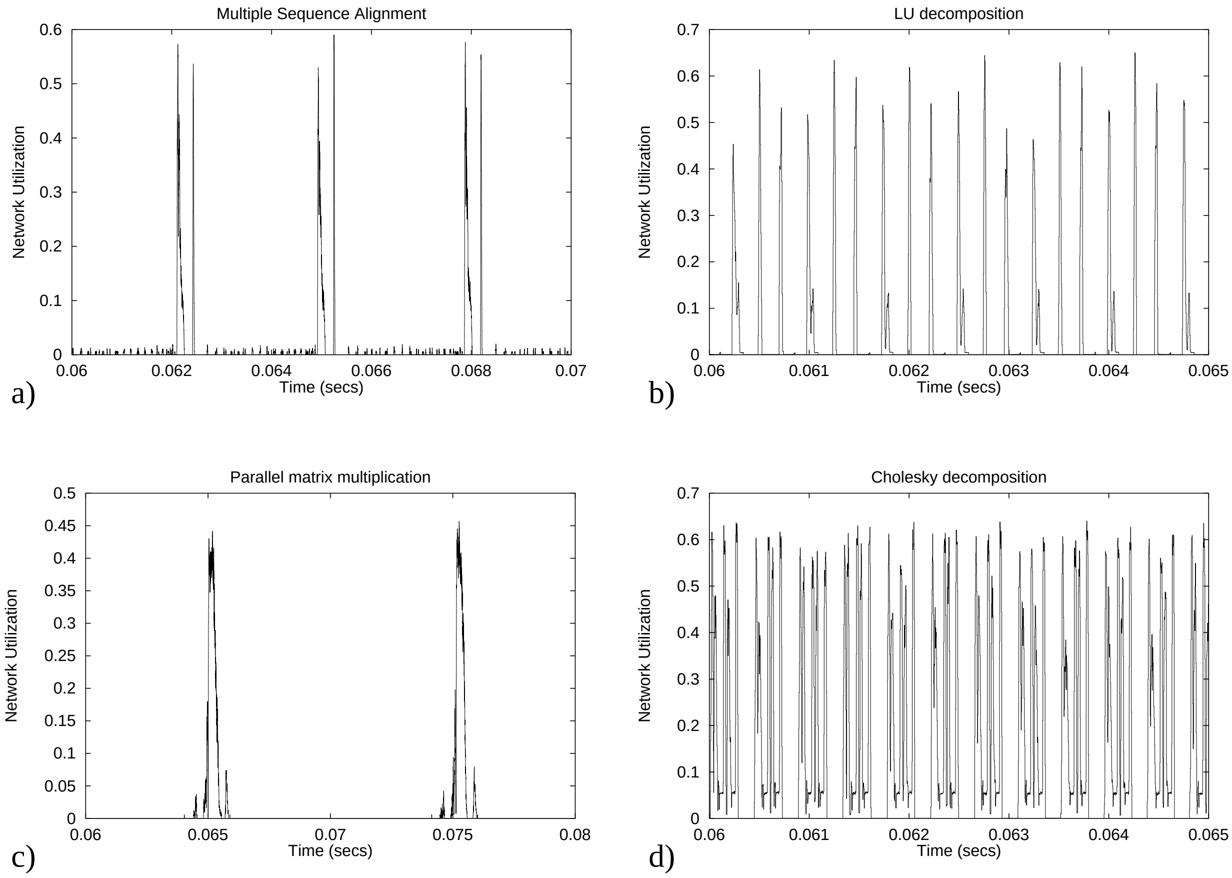

Figure 1: Network Utilization in Scientific Parallel Programs.

(where processors make decisions based solely on their local status and a limited view of the remote status) to more outward-looking or global scheduling algorithms. Third, non-blocking, one-sided communication primitives decouple communication and synchronization, thus allowing the communication pattern to be scheduled with additional degrees of freedom.

This approach 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.

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 buffered coscheduling can be tolerant to the potentially high latencies that can be introduced in a kernel call or in the initialization of the network interface card (NIC) that can reside on a slow I/O bus. In addition to amortizing communication and scheduling overhead, we can also implement zero-copy (or low-copy, if we desire fault-tolerant communication) communication. As a result, our approach to communication buffering can achieve performance comparable to user-level network interfaces (i.e., OS-bypass protocols) [3] 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. The strobing mechanism performs an optimized total-exchange of control information (which we call heartbeat) and triggers the downloading of any buffered packets into the network.

Figure 2 shows how computation and communication can be scheduled over a generic processor. At the beginning of the heartbeat, t_0, the kernel downloads control packets into the network for a total exchange. During the execution of the barrier synchronization, another user process gains control of the processor; and at the end of the barrier synchronization, the kernel schedules the pending communication, accumulated in the previous time-slice (before t_0), to be delivered in the current time-slice $[t_0, t_2]$. From the control information exchanged between t_0 and t_1, the processor will know (at t_1) 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. It is worth noting that the potentially high overhead of the barrier synchronization is simply removed from the critical path by running another process. Thus, we can tolerate the latency of a global exchange of information without experiencing performance degradation.

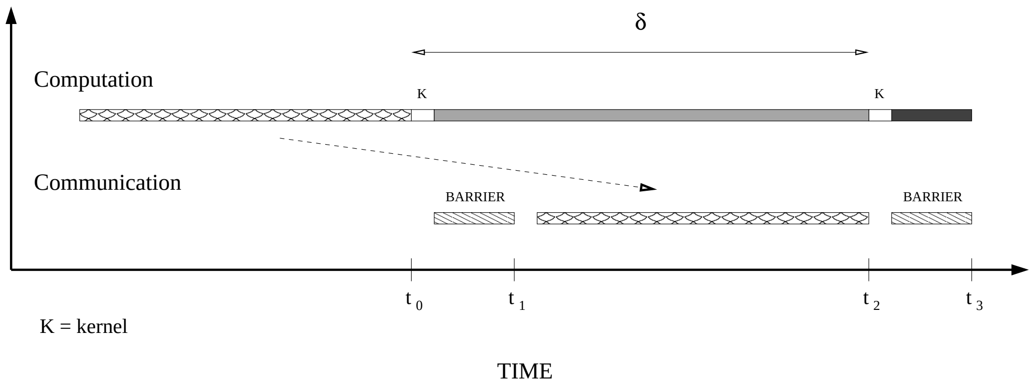

Figure 2: Scheduling Computation and Communication. Communication accumulated in the time-slice up to t_0 is downloaded into the network between t_1 and t_2 (after the barrier synchronization). $\delta \equiv$ length of a time-slice $= t_2 - t_0$.

This strategy can be easily extended to deal with space-sharing where different regions run different sets of programs [14]. In this case too, all the different regions are synchronized by the same heartbeat.

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 or one-sided communication.

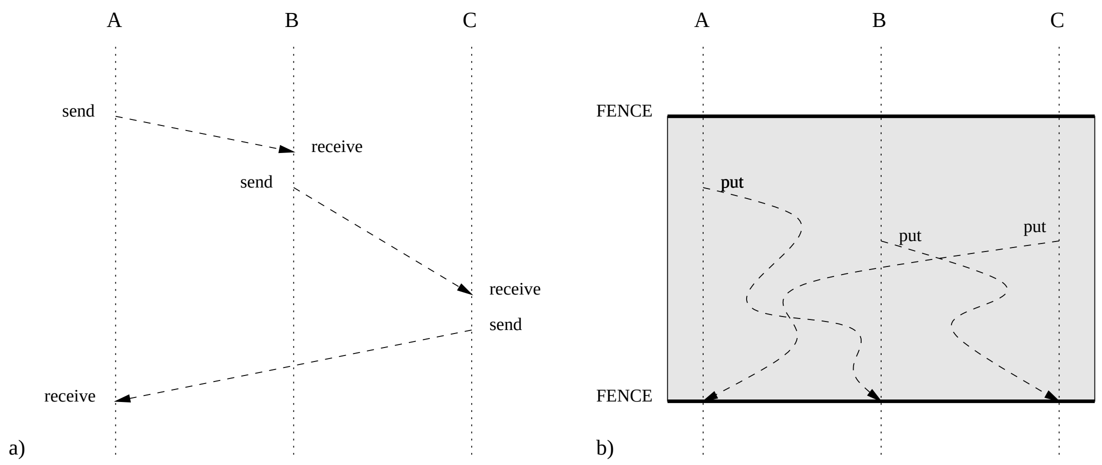

Figure 3: (a) Message Passing (b) One-Sided Communication.

As Figure 3 shows, 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. Arrows between processes represent communication between a sender and a receiver. In Figure 3(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 3(b), the same communication pattern is delimited by two *barriers* and includes 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. Lastly, in contrast to explicit coscheduling, 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 [28]. Although the buffering and strobing mechanisms alone improve parallel program performance, transforming by themselves 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[1] can be routed with predictable time [10, 24]. 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 [24, 5, 23].

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.

The *BSP* model is also beneficial for fault tolerance[2] Fault tolerance can be enhanced by exploiting the synchronization points at the end of a time-slice: we can take a snapshot of the whole machine and checkpoint its status.

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

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

4 Experimental Results

Our preliminary experimental results include a working implementation of a representative subset of MPI-2 on a detailed simulation model [25]. The simulation environment includes a standard version of MPI-2 and a multitasking one, that implements the main features of buffered coscheduling. Because the design space of our problem is too large to explore exhaustively, we fix the workload and system characteristics.

4.1 Characteristics of the Synthetic Workloads

The workloads used consist of a collection of single-program multiple-data (SPMD) parallel jobs, similar to those reported in [6], which alternate phases of purely local computation with interprocess communication. A parallel job generated by one of such programs consists of a group of P processes and each process is mapped on a processor throughout the 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 optional sequence of communication events terminated by a closing barrier. We consider three communication patterns: *Barrier*, *News Transpose*. *Barrier* consists of only the closing barrier and thus contains no additional dependencies. This workload can be used to analyze how buffered coscheduling responds to load imbalance. The other two patterns consist of a sequence of remote writes. The communication pattern generated by *News* is based on a stencil with a grid where each process exchange information with its four neighbors. This workload represents those applications that perform a domain decomposition of the data set and limit their communication pattern to a fixed set of partners. *Transpose* is a communication-intensive workload that emulates the communication pattern generated by the FFT transpose algorithm [12], where each process accesses data on all other processes.

We consider three parallel jobs with the same computation granularity, load-imbalance and communication pattern arriving at the same time in the system. 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 the experimental evaluation is called SMART (Simulator of Massive ARchitectures and Topologies) [21], 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 [27]. In particular, it models a two-level cache hierarchy with a write-back L1 policy and non-blocking caches. We assume a processor speed of 500 MHz. In the experiments we will consider two networks with 32 processing nodes, representative of two different architectural solutions.

The first network is a 5-dimensional cube topology with performance characteristics similar to those of Myrinet routing and network cards [4]. This network features a one-way data rate of about 1 Gbit/s and a base network latency of less than a μs.

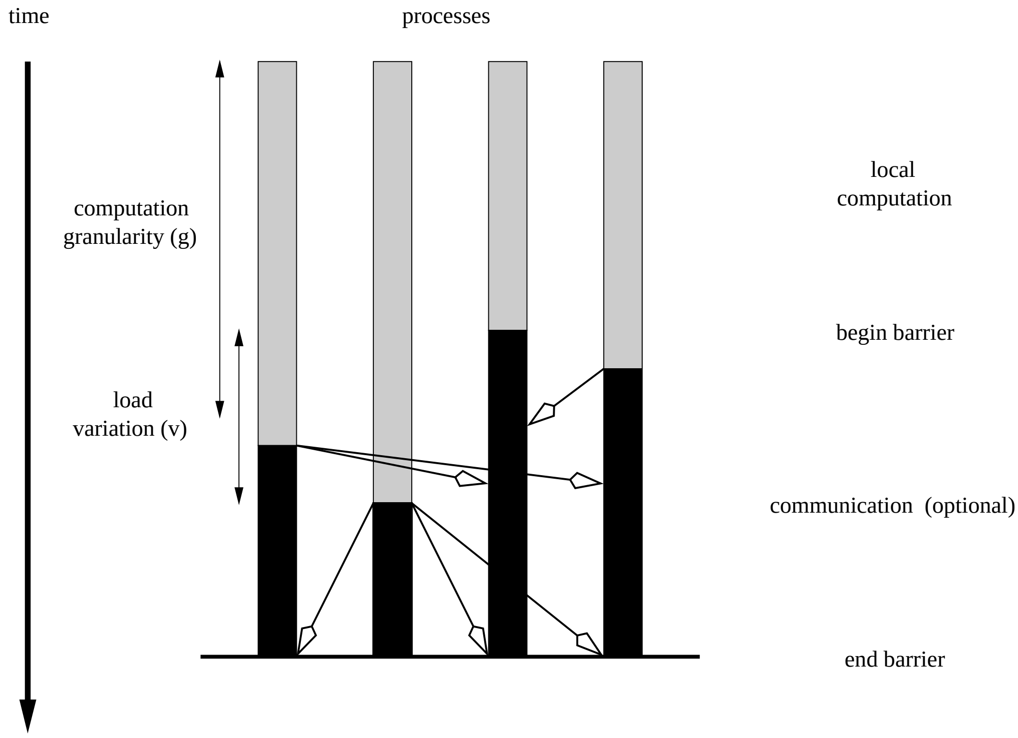

Figure 4: Each process of a parallel job executes on a separate processor and alternates between computation and communication. Processes compute for a mean time g before executing the closing barrier of the communication phase. The variation in computation across processes is uniformly distributed in $(0, v)$.

The second network is based on 32-port, 100 Mbit/s switch, a popular solution due its attractive performance/price ratio. An example are the Intel Express switches[3]. This network displays latencies of less than 10 μs.

The simulator models at register level the congestion inside the network, at the network interface and the routing and flow control protocols. The run-time support running on this simulated platform includes a standard version of a significant subset of MPI-2 that runs each job in dedicated mode and a multitasking version of the same subset that performs buffered coscheduling as outlined in section 3. The standard version uses an aggressive user-level messaging system [2, 17, 19, 26] which introduces very little overhead and exposes the full communication capability provided by the hardware.

4.3 Sensitivity Analysis

Figures 5 and 6 illustrate the communication and computation characteristics of our synthetic benchmarks as a function of the communication pattern, granularity, load imbalance, and time-slice duration. 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, in the Myrinet-based interconnection network, we consider time-slices of 500 μs,

[3]See http://www.intel.com/network/products/express_switches.htm.

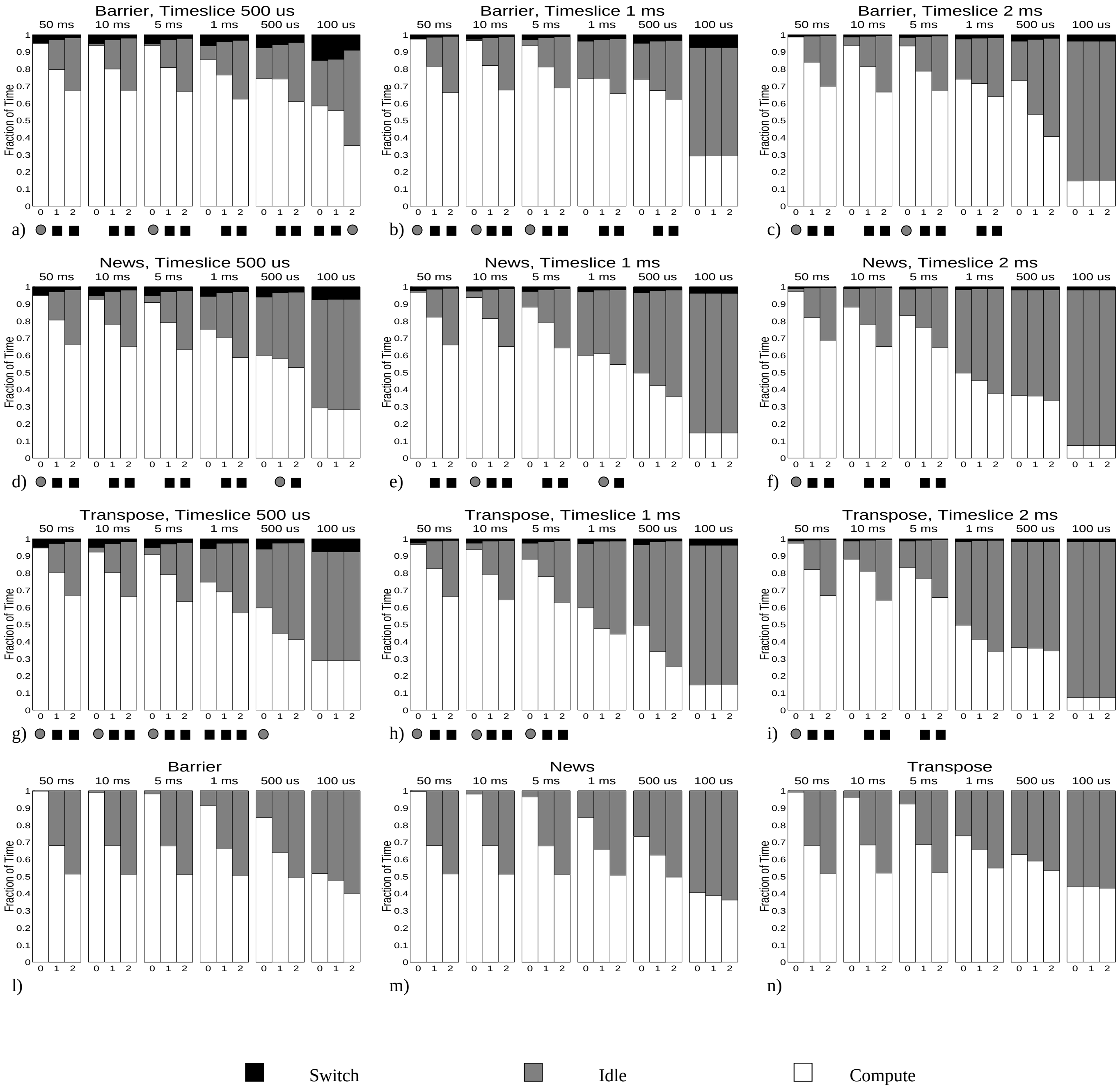

Figure 5: Execution characteristics as a function of computation granularity, load imbalance, time-slice length for the Myrinet-based interconnection network. A black square under a bar highlights the configurations where the multitasking approach gets better resource utilization than the standard approach, and a circle indicates the configurations where the performance loss is within 5%.

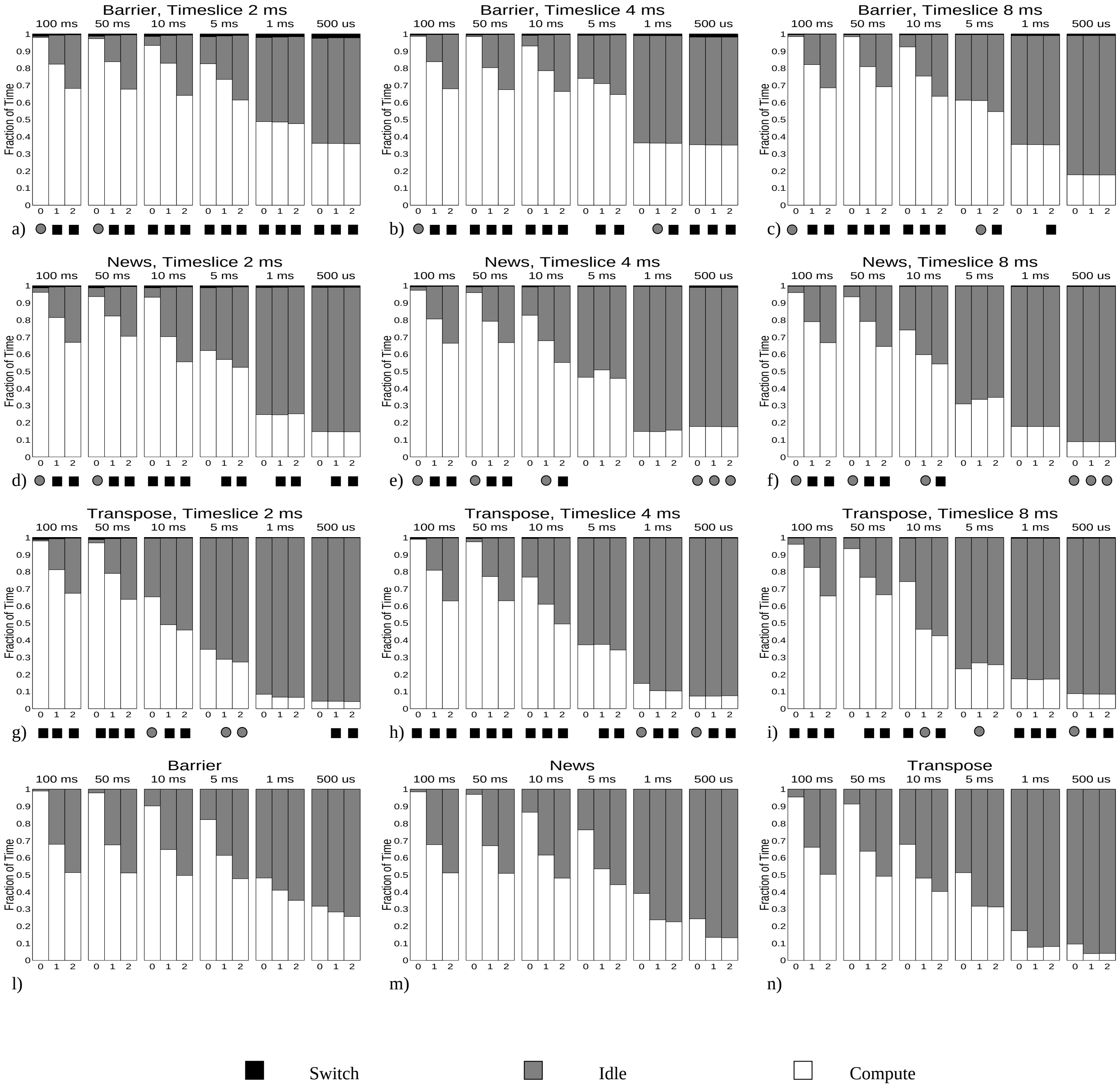

Figure 6: Execution characteristics as a function of computation granularity, load imbalance, time-slice length for the switch-based network. A black square under a bar highlights the configurations where the multitasking approach gets better resource utilization than the standard approach, and a circle indicates the configurations where the performance loss is within 5%.

1 and 2 ms. Due to larger communication overhead and lower bandwidth, in the switch-based network the time-slices are 2, 4 and 8 ms. In both cases the context switch penalty is 25 μs. For each of these alternatives, we considers six groups of three bars. Each group has the same computation 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).

Figures 5 (l)-(n) and 6 (l)-(n) show the breakdown for the *Barrier*, *News* and *Transpose* workloads when they are run in dedicated mode on the MPI-2 run-time support. For Figures 5 (a)-(i) and 6 (a)-(i) a black square under a bar highlights the configurations where buffered coscheduling gets better resource utilization than MPI-2 user-level communication, and a circle indicates the configurations where the performance loss of buffered coscheduling is within 5%.

By examining the breakdowns of each bar, we can see several important trends. As the load imbalance of the program increases (i.e., moving to the right within each group of three bars with the same computational granularity), the idle time increases. The time-slice length is a critical parameter in determining overall system performance. A short time-slice can achieve very good 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 cannot reduce the load imbalance, in particular when there are fine-grained computations.

In Figures 5 (a), (d) and (g), we see that with a Myrinet-based network, using a relatively small time-slice, our multitasking environment produces higher processor utilization than when a single job runs in a dedicated environment in more than 50% of the cases. For most of the other cases (i.e., $v = 0$ or perfectly balanced jobs), running a single job results in *marginally* better performance because buffered coscheduling must "pay" the context-switch penalty without improving the load balance because the load is already balanced. On the other hand, in the presence of load imbalance, job multitasking can smooth the differences in load, resulting in both higher processor and network utilization. A less powerful interconnection network amplifies the positive characteristics of buffered coscheduling. In Figures 5 (a), (d) and (g), we see that with the switch-based interconnection network, buffered coscheduling outperforms the basic approach in 16 configurations out of 18 with *Barrier*, 13 out 18 with *News* and 10 out 18 with *Transpose*. In this last case, the performance of buffered coscheduling can be improved by increasing the time-slice. In fact, when the time-slice is 4 ms, the number of improved configurations is 15 out of 18.

As a final note, our preliminary experimental results do not account for 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.

4.4 Job Scheduling

In the experiments shown above we considered three jobs of the same kind (same computational grain size, variance and communication pattern). We now extend our analysis to workloads of multiple jobs with different communication patterns and computational granularity. The mixed workloads are generated by using two classes of jobs, BAR(g_b,v_b) (*Barrier* with a computational grain size of g_b ms and variance of v_b ms) and TRA(g_t,v_t) (*Transpose* with a computational grain size of g_t ms and variance of v_t ms). We consider five workloads: the first two with three jobs, and the last three with four jobs, as shown in Table 1. The number of iterations for each job is adjusted so that the total completion time for each workload in a dedicated environment is 100 seconds and so that each job takes approximately the same time to run, e.g., each of the three jobs in workloads 1 and 2 execute in 33.3 seconds, respectively, and each of the four jobs in workloads 3, 4, and 5 execute in 25 seconds, respectively. For each network, we also consider a distinct pair of jobs: In the Myrinet-based network we set (g_b,g_v) to (50 ms, 50 ms) and (g_t, g_v) to (1 ms, 2 ms), while in the switch-based network (g_b,g_v) is (100 ms, 100 ms) and (g_b,g_v) is (5 ms, 10 ms). In both cases, the first job is relatively computation intensive with some load imbalance while the second job is communication intensive with high load imbalance.

Workload	Jobs in Workload	Communication
1	BAR(g_b,g_v), TRA(g_t,g_v), TRA(g_t,g_v)	(low,hi,hi)
2	BAR(g_b,g_v), BAR(g_b,g_v), TRA(g_t,g_v)	(low,low,hi)
3	BAR(g_b,g_v), TRA(g_t,g_v), TRA(g_t,g_v), TRA(g_t,g_v)	(low,hi,hi,hi)
4	BAR(g_b,g_v), BAR(g_b,g_v), TRA(g_t,g_v), TRA(g_t,g_v)	(low,low,hi,hi)
5	BAR(g_b,g_v), BAR(g_b,g_v), BAR(g_b,g_v), TRA(g_t,g_v)	(low,low,low,hi)

Table 1: Mixed workloads with three and four jobs. BAR(g_b,g_v) identifies a *Barrier* with a grain size of g_b ms and a variance of v_b ms. TRA(g_t,g_v) is a *Transpose* with g_t ms and v_t ms. The first column indicates the job id, the second column shows the workload composition and the third column gives a quick overview of the communication intensities of each job in the workload. *low* and *hi* indicate, respectively, low and high communication intensity.

To determine the impact of local process scheduling, we also considered six scheduling algorithms. The first two use local information only. *Round Robin* (ROUND) is a simple local scheduler based on a FIFO ready queue. *Stride scheduling* (STRIDE) is a credit-based algorithm that tries to give fair processor allocation; resources (the processors in our case) are allocated to competing processes in proportion of the number of tickets they hold [29].

The remaining four scheduling algorithm use the global knowledge provided by buffered coscheduling through the total exchange of information that takes place during each time-slice. *Blocked Scheduling* (BLOCKED) gives priority to the job with the maximum number of processes blocked on a synchronization primitive (e.g., a barrier synchronization or a fence). The rationale is that the active processes in this job are potentially delaying all the remaining processes, so they should receive a preferential treatment. *Time-slice Scheduling* (SLICE) applies the same credit-based algorithm of Stride scheduling at the job level rather than at process level. Each job is given a certain amount of tickets and the local scheduling decision is determined by the global state of the jobs. *Fair Blocked Scheduling* (F_BLOCK) tries to improve the fairness of Blocked Scheduling by periodically boosting the priority of the jobs at the beginning of each time-slice. For example, let us assume that the number of jobs running in the system is *jobs* and that the time-slice id is *time_id*. At the beginning of the time-slice *time_id*, the process belonging to the job *job_id*

$$job\_id \;=\; time\_id \; MOD \;\; jobs,$$

sees its priority boosted. This guarantees that each job/process has a fixed slot assigned to it at regular intervals. The same scheduling strategy can be used to extend the Time-slice scheduling and obtain the *Fair Time-slice Scheduling*

(F_SLICE). The fair versions of the algorithms can be used to deliver quality of service (QoS) in time-critical or multimedia applications.

	Workloads, Myrinet				
	1	2	3	4	5
ROUND	78.5	89.3	68.7	74.4	79.8
STRIDE	74.1	77.1	65.2	69.8	77.9
BLOCK	75.4	79.7	64.7	65.9	81.2
SLICE	74.0	77.2	63.8	70.6	75.3
F_BLOCK	75.8	78.8	66.9	66.9	74.8
F_SLICE	74.2	78.5	66.6	71.7	75.0

Table 2: Execution times, expressed in seconds, of the workloads on the Myrinet-based interconnection network, using different scheduling strategies.

	Workloads, Switch				
	1	2	3	4	5
ROUND	89.5	79.4	86.6	74.0	73.8
STRIDE	87.0	74.5	85.3	68.4	69.4
BLOCK	87.1	75.4	68.1	68.1	76.3
SLICE	87.1	69.9	70.8	67.6	70.7
F_BLOCK	89.3	77.1	70.5	73.2	68.2
F_SLICE	89.4	78.3	67.3	69.1	69.7

Table 3: Execution times, expressed in seconds, of the workloads on the switch-like interconnection network, using different scheduling strategies.

The preliminary experimental results reported in Tables 2 and 3 compare the scheduling algorithms using the workloads shown in Table 1. From these results we can see that buffered coscheduling is relatively insensitive to the scheduling policy. This implies that buffered coscheduling works well across all classes of applications regardless of the scheduling policy.

Tables 2 and 3 also show that the global execution time is less than 100 seconds in all cases. This implies that buffered coscheduling can significantly improve the resource (processor) utilization while *also* improving overall system throughput. This is in stark contrast to [18] where all the optimized coscheduling algorithms produce (i.e., tradeoff) reduced system throughput in order to get better resource utilization and response time.

For example in Table 2, we see that with workload 3, the most communication intensive, the completion time is 35% lower, on average. More jobs give more opportunities to globally optimize the resource (as long as they do not require extra paging). Though there is not a scheduling algorithm that clearly outperforms all the others, we can see that Round Robin lags behind in most cases. The best performance is usually provided by Blocked and Time-slice scheduling. Surprisingly, Stride scheduling also provides comparable performance using information available locally. The fair versions of Blocked and Time-slice scheduling are in the same performance range of the corresponding basic algorithms. Tables 4 and 5 provide more insight on the individual completion times of each job for workload 3.

	Workload 3, Myrinet				
	low	hi	hi	hi	E
ROUND	67.5	90.4	90.9	91.7	85.1
STRIDE	76.3	86.3	86.8	87.0	84.1
BLOCK	73.8	86.3	85.2	86.1	82.8
SLICE	76.2	84.4	85.1	84.8	82.6
F_BLOCK	64.4	86.1	89.2	88.0	81.9
F_SLICE	72.0	88.8	88.2	88.4	84.3

Table 4: Individual completion times for workload 3 on the Myrinet-based network.

	Workload 3, Switch				
	low	hi	hi	hi	E
ROUND	38.0	61.0	73.2	86.6	64.7
STRIDE	36.1	60.2	72.2	85.3	63.4
BLOCK	41.2	66.2	66.0	68.1	60.3
SLICE	40.8	67.2	70.8	69.9	62.1
F_BLOCK	38.0	68.7	69.9	70.5	61.7
F_SLICE	38.6	66.4	65.6	67.3	59.4

Table 5: Individual completion times for workload 3 on the switch-based network.

The results reported in [18] consider mixed workloads with little load imbalance and a limited percentage of execution time performing communication (i.e., $< 26\%$). For their workloads the slowdown with respect to the sequential case varies between 30% and 180% in the best case. In contrast, in our workloads, the fraction of time spent in communication ranges up to 95% (*Transpose* with a computational grain size of 100 μs and a switch-based interconnection network). Our experimental results show that in a large majority of cases, we experience *speedup* rather than slowdown. And even in the remaining cases, the slowdown is often less than 5%. Though the results are not directly comparable, the performance difference between buffered coscheduling and those reported in [18] is significant enough to lead us to believe that buffered coscheduling can bridge the gap between the "faster response time but lower throughput" of coscheduling and the "slower response time but higher throughput" of batch scheduling.

5 Conclusion and Future Work

In this paper we have presented buffered coscheduling, a new methodology to multitask parallel jobs on a parallel computer. The methodology addresses the main limitation of explicit coscheduling — the high latency needed to perform a global context switch — while keeping the main advantage of dynamic coscheduling — the possibility of overlapping computation, communication and I/O of different jobs. Also, it provides a simple framework to increase the resource utilization, simplify the design of the run time support, increase the faults tolerance and perform effective global optimizations.

We initially addressed the complexity of a huge design space using three families of syntethic workloads and considering two distinct interconnection networks. The preliminary experimental results reported in this paper show that buffered coscheduling can provide better resource utilization in the vast majority of cases, in particular 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 with real applications rather than synthetic workloads and to implement in a Linux cluster a multitasking version of MPI-2.

References

[1] Andrea C. Arpaci-Dusseau, David Culler, and Alan 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. Ruhl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11), November 1998.

[3] Raul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawick, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995.

[5] Douglas C. Burger and David 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.

[6] Andrea C. Dusseau, Remzi H. Arpaci, and David 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.

[7] Dror G. Feitelson and Morris 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.

[8] Dror G. Feitelson and Larry 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.

[9] Dror G. Feitelson and Larry Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

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

[11] 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.

[12] Anshul Gupta and Vipin Kumar. The Scalability of FFT on Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993.

[13] Adolfy Hoisie, Olaf Lubeck, and Harvey 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.

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

[15] Vijay Karamcheti and Andrew A. Chien. Do Faster Routers Imply Faster Communication? In *First International Workshop, PCRCW'94*, volume 853 of *LNCS*, pages 1–15, Seattle, Washington, USA, May 1994.

[16] Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry 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.

[17] R. Minnich, D. Burns, and F. Hady. The Memory-Integrate Network Interface. *IEEE Micro*, February 1995.

[18] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita 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.

[19] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2):60–73, April-June 1997.

[20] William E. Weihl Patrick Sobalvarro, Scott Pakin and Andrew 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.

[21] F. Petrini and M. Vanneschi. SMART: A Simulator of Massive ARchitectures and Topologies. In *Proceedings of the International Conference on Parallel and Distributed Systems Euro-PDS'97*, June 1997.

[22] Fabrizio Petrini. Network Performance with Distributed Memory Scientific Applications. Submitted to the Journal of Parallel and Distributed Computing, September 1998.

[23] Fabrizio Petrini. Total-Exchange on Wormhole k-ary n-cubes with Adaptive Routing. In *Proceedings of the 12th International Parallel Processing Symposium, IPPS'98*, Orlando, FL, March 1998.

[24] Fabrizio Petrini and Marco Vanneschi. Efficient Personalized Communication on Wormhole Networks. In *The 1997 International Conference on Parallel Architectures and Compilation Techniques, PACT'97*, San Francisco, CA, November 1997.

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

[26] I. R. Philp and Y. Liong. The Scheduled Transfer (ST) Protocol. In *Proceedings of Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, January 1999.

[27] Tom Shanley. *Pentium Pro and Pentium II System Architecture.* Addison-Wesley, March 1998.

[28] D. B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Journal of Scientific Programming*, 1998.

[29] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD thesis, Massachusetts Institute of Technology, September 1995.