Time-Sharing Parallel Jobs in the Presence of Multiple Resource Requirements

Fabrizio Petrini and Wu-chun Feng

Computing, Information, & Communications Division
Los Alamos National Laboratory, NM 87544, USA,
fabrizio@lanl.gov, feng@lanl.gov

Abstract. Buffered coscheduling is a new methodology that can substantially increase resource utilization, improve response time, and simplify the development of the run-time support in a parallel machine. In this paper, we provide an in-depth analysis of three important aspects of the proposed methodology: the impact of the communication pattern and type of synchronization, the impact of memory constraints, and the processor utilization.
The experimental results show that if jobs use non-blocking or collective-communication patterns, the response time becomes largely insensitive to the job communication pattern. Using a simple job access policy, we also demonstrate the robustness of buffered coscheduling in the presence of memory constraints. Overall, buffered coscheduling generally outperforms backfilling and backfilling gang scheduling with respect to response time, wait time, run-time slowdown, and processor utilization.

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

1 Introduction

The scheduling of parallel jobs 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: the workload, the parallel programming language, the operating system (OS), and the machine architecture. The importance of job scheduling strategies stems from the impact that they can have on the resource utilization and the response time of the system.

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, 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 software communication overhead and the scheduling overhead to wake up a sleeping process dominate the communication time on most parallel machines [14].

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: *gang scheduling* (GS), *local scheduling* (LS) and *implicit or dynamic coscheduling* (DCS).

On the one end of the spectrum, GS [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, these straightforward implementations are neither scalable nor reliable. Furthermore, GS 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 [17].

At the other end of the spectrum is LS, where each processor independently schedules its processes. This is an attractive time-sharing option due to its ease of construction. However, the performance of fine-grained communication jobs can be orders of magnitude worse than with GS because the scheduling is not coordinated across processors [11].

An intermediate approach developed at UC Berkeley and MIT is DCS [1] [19] [4] [25]. With DCS, 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 probably send more messages in the near future. The main drawbacks of dynamic coscheduling include the high overhead of generating interrupts upon message arrival and the limited vision of the status of the system that is based on speculative information. Some aspects of these limitations are 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.

We recently proposed a new approach to job multitasking, called *buffered coscheduling* (BCS) [6]. BCS shows promise in integrating the positive aspects of GS, e.g., global coordination of jobs, along with positive aspects of DCS, e.g., increased resource utilization obtained by overlapping computation and communication of different jobs. The benefits of BCS include higher throughput, dramatic simplification of run-time support, reduced communication overhead, efficient global implementation of flow-control strategies and fault-tolerant protocols, and accurate performance modeling. Here, we focus on the performance of BCS in the presence of memory constraints.

Like DCS, BCS must address a couple of important problems. A first problem is the impact of the memory hierarchy: All the benefits obtained with job multitasking can be wiped out if the memory requirements of multiple jobs exceed the physical memory available and overflow in the swap space. Secondary memory can be orders of magnitude slower. A second problem is the impact of the type of the job communication and synchronization on the overall throughput. This problem leads to another closely related problem: the choice of the time-slice length. While a long time-slice can hide the overhead and increase the scalability of BCS, it can also increase the processor idle time due to blocking communication.

In this paper, we analyze the above problems with a detailed simulation model driven by a real workload drawn from an actual supercomputing environment at Lawrence Livermore National Labs. By considering a simple job-scheduling algorithm that limits the access into the system of those jobs that exceed the memory requirements, we evaluate the system response time and utilization under various types of workloads and system parameters.

The rest of the paper is organized as follows. Section 2 briefly reviews BCS. Section 3 describes the job access policy that takes into consideration the memory requirements, Section 4 the experimental framework and Section 5 the results of the simulations. Some considerations on the potential advantages on the development of system-level and user-level software are listed in Section 6, the relations between BCS and the Bulk-Synchronous Parallel model of parallel computation are described in Section 7, followed by a conclusion in Section 8.

2 Buffered Coscheduling

To implement job multitasking, BCS relies on two 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 in order to move from isolated scheduling algorithms [1] (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. An important characteristic of BCS is that, 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.

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 GS, while maintaining the overlapping of computation and communication provided by DCS.

2.1 Communication Buffering

Rather than incurring communication and scheduling overhead on a per-message basis, BCS accumulates the messages generated by each process and tries to 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 BCS 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.

2.2 Strobing Heartbeats

Virtually all the existing research in parallel job scheduling use isolated algorithms, which speculatively make scheduling decisions based on a limited knowledge of the status of the machine, rather than algorithms which use non-isolated (or even global) knowledge. 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. Our goal is to synchronize the processors of the parallel machine at the end of a time-slice in order to perform a total exchange of information regarding their status. To amortize the overhead, all the communication operations are buffered and executed at the end of the time-slice. The strobing mechanism performs an optimized total-exchange of control information (which we call heartbeat or strobe) and triggers the downloading of any buffered packets into the network. At the start of the heartbeat, each processor downloads a personalized broadcast 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 will enter a phase where its kernel will download any buffered packets. 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 1 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 heartbeat, another user process gains control of the processor; and at the end of the heartbeat, the kernel schedules the pending communication, accumulated in the previous time-slices (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 strobing algorithm 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.

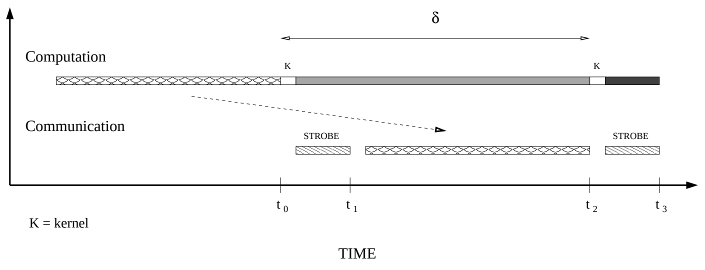

Fig. 1. Scheduling Computation and Communication. Communication accumulated in the time-slice up to t_0 is downloaded into the network between t_1 and t_2 (after the heart beat). $\delta \equiv$ length of a time-slice $= t_2 - t_0$.

3 Job Access Control

Scheduling parallel jobs by sharing processors not only spatially but also temporally provides an extra degree of flexibility and a considerable performance advantage. Unfortunately, this advantage can be limited by multiple resource requirements, e.g., memory hierarchy requirements. If the jobs mapped on a processing node exceed the physical memory available and use the virtual memory, the advantages of job multitasking can be nullified.

In order to avoid such problem we consider a very simple job access control policy, which allows jobs into the system only if their memory requirements do not exceed the physical memory available. For instance, Figure 2 shows the Ousterhout matrix of an 8-processor system with a multiprogramming level of three and 512 MB of physical memory per processor P_i. Job J_5 requires 2 processors and 256 MB of memory per processor. Thus, it can only be mapped onto two of the four two-processor slots available due to memory constraints.

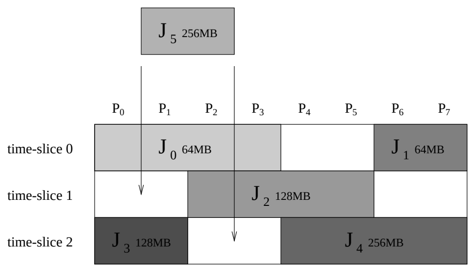

Fig. 2. Ousterhout Matrix of an 8-Processor System with 512-MB Memory/Processor and Multiprogramming Level of 3.

In our experiments, we combine the above access control policy with an aggressive backfill heuristic [24], which selects any job from the ready queue that does not interfere with the expected start time of the first blocked job. As shown in [27], this technique, when used with GS, can provide improvements over a wide spectrum of performance criteria. However, this greedy method does not look at the additional resource requirements of the jobs in the ready queue or the current state of the system resource loads, thus leaving room for future improvements.

GS can re-use some of the unused slots in the Ousterhout matrix if a job assigned to a given time-slice can atomically fit into one or more empty slots in another time-slice. This is the case of jobs J_1 and J_3 in Figure 2, which can be run on the two slots available in time-slice 1, as shown in Figure 3.

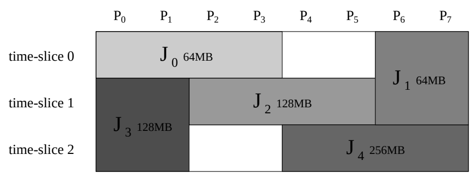

Fig. 3. Empty slot utilization with GS

While GS cannot fill in the two unused slots in the time-slices 0 and 2 with job J_2, BCS can potentially use their processing time, because the grain size of the resource allocation is the process and not the entire job. The communication pattern of the jobs, the local process scheduling algorithms, and many other factors can influence how the resources made available by the empty slots can be used by different jobs.

4 Experimental Framework

Before presenting the experimental results, we provide details on our simulation platform, the workloads used to drive the simulator, and the metrics of interest.

4.1 Simulation Model

In order to efficiently simulate and analyze different job scheduling strategies for parallel computers in depth, we developed a novel simulator called the *Job Scheduling Simulator* (JSS). With JSS, the user can explore the Cartesian product generated by different dimensions of the design space. A first dimension is machine scheduling: JSS provides space sharing and two basic forms of time sharing — gang scheduling (GS) and buffered coscheduling (BCS). A second dimension is the selection algorithm of the ready-jobs queue. Jobs can be selected in FCFS (First Come First Served) order or backfilled using a *conservative* or an *aggressive* policy. Conservative backfilling searches the ready queue for jobs that can be scheduled immediately, with the constraints that these jobs cannot interfere with the expected start time of the jobs which come before them in the ready queue. Aggressive backfilling is a weaker version of conservative backfilling, which selects any job from the queue which does not interfere with the expected start time of the first job in the ready queue. Both conservative and aggressive backfilling can dramatically improve the overall machine utilization and response time over FCFS but require a reasonably good estimate of the job run-time.

Both GS and BCS can have a parametric multiprogramming level (MPL) and times-slice length and can use the job access control policy described in Section 3. With GS, the user can also set the delay associated with job context-switch at the end of each time-slice.

In our BCS implementation, the user can define the system parameters as the process context-switch penalty, communication bandwidth between processors, and the algorithms to globally schedule the communication pattern. In order to explore how the various aspects of computation and communication influence the overall performance of BCS, JSS provides an API, composed of a limited but representative subset of MPI, that includes blocking and non-blocking communication primitives and synchronization primitives. The current implementation of JSS abstracts the main characteristics of each job using four parameters $\langle g, v, comm, sync \rangle$, where g represents the computational grain size, v the load imbalance, *comm* the communication pattern, and *sync* the type of synchronization. A parallel job consists of a group of P processes, and each process is mapped on a processor throughout the execution. Processes alternate phases of purely local computation with interprocess communication, as shown in Figure 4. Each process 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. By varying v, we change the degree of load-imbalance across processors. The communication phase consists of an optional sequence of communication events. The parameter *comm* defines one of the three communication patterns: *Barrier*, *News* and *Transpose*. *Barrier* does not perform any communication and can be used to analyze how buffered coscheduling responds to load imbalance. The other two patterns consist of a sequence of point-to-point communications. The communication pattern generated by *News* is based on a stencil with a grid where each process exchanges information with its four neighbors. This pattern 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 of all other processes. Finally, *sync* describes the type of synchronization in a job: we can have either blocking communication (B), where each point-to-point communication is implemented with blocking sends and receives or non-blocking communication (NB), where the communication primitives do not require an explicit handshake between sender and receiver and are terminated by a global barrier synchronization.

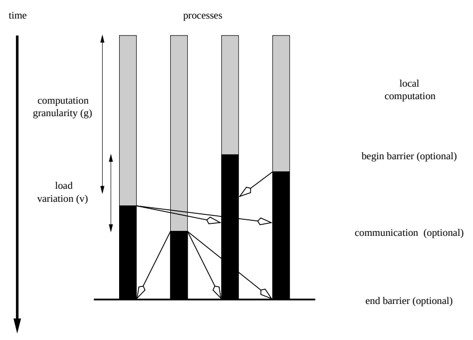

Fig. 4. Overlap of Computation and Communication

Parameter	Value
Processors	32
Main memory per processor	512 MB
Job context-switch (GS)	1 ms
Process context-switch (BCS)	100 μs
Message size (BCS)	4KB
Communication Bandwidth (BCS)	100 MB/s

Table 1. Experimental System Parameters and Values.

Table 1 describes some of the system parameters used during the experimental evaluation. We consider an architecture with 32 processors where each processor is equipped with 512 MB of main memory.

4.2 Workloads

A crucial aspect in the performance evaluation of job scheduling strategies is the availability of realistic workloads that can be represented with a compact mathematical formulation. Parallel workloads are often dispersive: job interarrival time distribution and job execution time distribution have a coefficient of variation that is greater than one, i.e., they are long tailed. These distributions can be fitted adequately with Hyper Erlang Distributions of Common Order [13]. Our experiments use a workload directly extracted from a real supercomputing environment, ASCI Blue-Pacific at Lawrence Livermore National Laboratory. Our modeling procedure involves the following steps.

1. The jobs are first grouped into classes, based on the number of processors they require. Each class is a bin in which the upper boundary is a power of two.
2. The original workload contains jobs varying in size from one to 256 processors. However, due to the large amount of details involved in the simulation of buffered coscheduling, we have limited ourselves to 32 processors, selecting jobs that fall within this limit. The resulting workload is a subset of the original workload and contains all the jobs that request up to 32 processors. It is worth noting that such a workload is extremely demanding, when run on a machine with only 32 processors.
3. We then model the inter-arrival time and the execution time distributions for each class through Hyper Erlang Distributions of Common Order.
4. Next we generate various synthetic workloads from the observed workload by multiplying the average job execution time by a *load factor* from 0.1 to 1.6 in steps of 0.1. For a fixed inter-arrival time, increasing job execution time typically increases resource utilization, until the system saturates. The load factor 1.0 identifies the observed workload.
5. Each job requires an amount of main memory which is exponentially distributed around a given mean value, which represents the maximum memory requirements over all processes belonging to a job.

When simulating buffered coscheduling, we need an extra degree of detail to characterize how computation, communication and synchronization are performed inside each job. Thus, the modeling procedure requires some extra steps.

1. Based on the workload characterization, we pick a job template for each job in a workload.
2. Based on the job template, we determine the computation and communication patterns of the job.

Table 2 outlines the five workload characterizations used in the experiments: each one is composed of three job templates, described using the notation defined in Section 5. Jobs in a workload can be assigned one of the three templates with equal probability. These characterizations display different communication and synchronization patterns. In the first one (workload 0) all the jobs perform an intensive communication pattern (Transpose) using blocking communication. The second workload uses the same communication pattern together with non-blocking communication. The same characteristics distinguish workloads 2 and 3. They use the same communication pattern, News, but a different type of synchronization. In the fifth workload, jobs do not perform any communication: the goal of this workload is to identify the impact of load imbalance.

4.3 Metrics

The experimental evaluation considers metrics that are important from both the system's and user's perspectives.

Workload	Job Template 0	Job Template 1	Job Template 2
0	$\langle 50, 25, Tra, B \rangle$	$\langle 100, 50, Tra, B \rangle$	$\langle 200, 100, Tra, B \rangle$
1	$\langle 50, 25, Tra, NB \rangle$	$\langle 100, 50, Tra, NB \rangle$	$\langle 200, 100, Tra, NB \rangle$
2	$\langle 50, 25, News, B \rangle$	$\langle 100, 50, News, B \rangle$	$\langle 100, 100, News, B \rangle$
3	$\langle 50, 25, News, NB \rangle$	$\langle 100, 50, News, NB \rangle$	$\langle 200, 100, News, NB \rangle$
4	$\langle 50, 25, Barrier, NB \rangle$	$\langle 100, 50, Barrier, NB \rangle$	$\langle 200, 100, Barrier, NB \rangle$

Table 2. Five Workloads: Each with an equal mix of three job classes. The job granularity and skew are expressed in ms.

- *Wait Time*: The time spent by a job waiting in the ready queue before it is scheduled.
- *Execution Time*: The actual job run time.
- *Response Time*: The sum between wait and execution time.
- *System Utilization*: The system utilization identifies the machine utilization at the job allocation level. Intuitively, the system utilization is the fraction of the scheduling matrix that is filled with jobs.
- *Processor Utilization*: The processor utilization is the fraction of time CPU spent is useful computation. It is worth noting that, in the general case, the processor utilization is always smaller than the system utilization, because the processors can be idle during the job execution.
- *Execution Time Slowdown*: The execution time slowdown is the ratio between the execution time and the job run time in a dedicated environment. The execution time slowdown is 1.0 with space sharing and a number larger than 1.0 in a time shared environment.

5 Experimental Results

The experimental results try to provide insight into three important aspects of buffered coscheduling: (1) the impact of the communication pattern and the time-slice length on the response time, (2) the impact of memory constraints with the job access control policy outlined in section 3 and the (3) the processor utilization. In all three cases we compare buffered coscheduling with aggressive backfilling (BF), a scheduling policy that can obtain excellent performance results with space sharing [24], and with backfilling gang scheduling (BGS), the extension of this technique to gang scheduling, recently proposed in [27].

5.1 Impact of Communication, Synchronization and Time-slice Length

The choice of the time-slice for BCS is the result of a compromise between competing factors. On the one hand, a large time-slice could easily hide the overhead associated with the strobing algorithm and the process context-switches, thus allowing the scalability of BCS to architectures with a large number of processors.

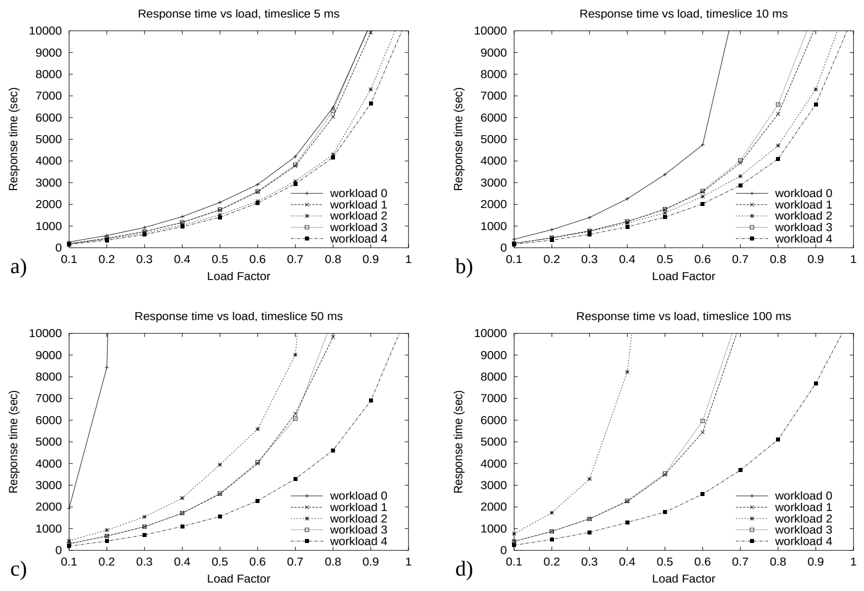

Fig. 5. Response time for various time-slices, communication and synchronization patterns. In all graphs the MPL is 3.

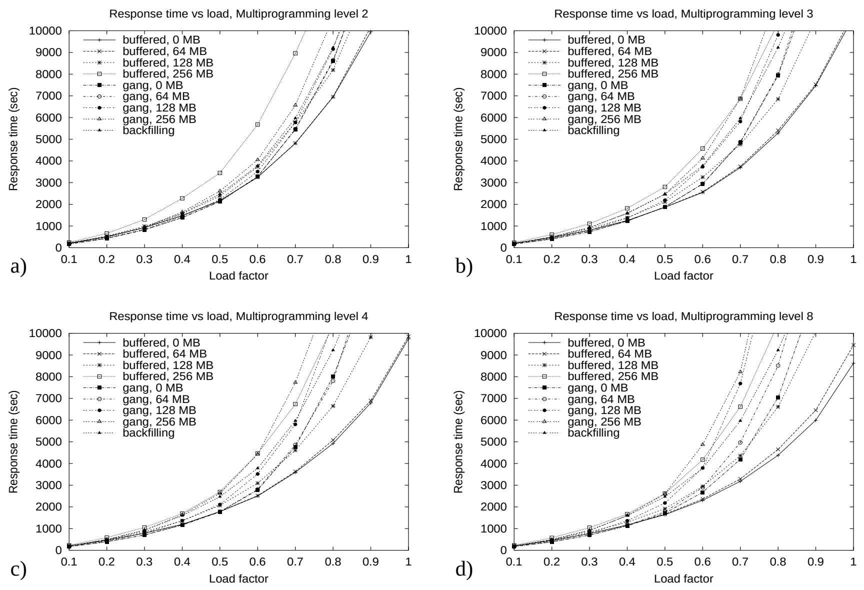

Fig. 6. Response time versus load for various MPLs. The graphs compare BCS with BF and BGS.

On the other hand, it increases the likelihood of having idle processors, due to blocking communication and global synchronization, thus limiting the potential increase in resource utilization.

Figure 5 shows how the response time is influenced by increasing the time-slice from 5 ms to 100 ms. All the experiments use a MPL equal to three. From the graphs, we can clearly see that workloads with a large amount of blocking communication (templates 0 and 2) can be efficiently supported only with small time-slices. This is particularly true for workload 0 which is extremely sensitive to the increase in the time-slice because it is communication intensive.

Looking at the graphs generated by templates 1 and 3 we can see that they almost overlap with all time-slices. These workloads share the same form of synchronization, obtained with a global barrier, though they have fairly different communication patterns. We explored this aspect in depth using many other communication patterns, workloads templates, number of processors, and architectural characteristics (not shown here for brevity), and we have found out that this is a strong property of BCS. With BCS, *the overall performance is relatively insensitive to the communication pattern when the communication is performed with non-blocking calls or, more generally, with a collective communication pattern.* The rationale behind this property is related to the fact that the run-time support cannot efficiently schedule blocking communication, while it can rearrange non-blocking primitives. This leads to a nearly optimal overlap between computation and communication when we use relative large MPLs. Also, there is an extra advantage in using collective communication patterns (e.g., broadcasts, scatter & gather, multicasts) because the information provided by the communication pattern can be directly passed to the run-time support, which can thus perform effective global optimizations. This is not true in the general case; in fact, many parallel applications possess a well defined communication structure that is lost in the compilation process (e.g., because it is mapped in an unstructured communication graph of blocking calls).

5.2 Impact of Memory Constraints

This section analyzes the machine response time, the wait time, and execution time slowdown in conjunction with the memory-aware job scheduling policy described in Section 3. In all experiments we use the workload template number 3 and we consider workloads with increasing average memory requirements, ranging from 0 MB (i.e., no memory constraints), to 256 MB, half the size of physical memory available on each processor.

From Figure 6 we can draw the following considerations:

- BCS outperforms GS in all configurations. This is more pronounced at higher loads, because BCS can overlap computation with communication and can re-use computing resources at the process-level granularity rather than at the job level, as shown in Section 3.
- There is no penalty in using an arbitrarily large MPL with BCS. For a given average memory requirement, the system converges to a given state and does

not experience any degradation when we further increase the MPL. That state is mainly determined by the ratio between the job average memory requirements and the actual physical memory available.

– When the memory requirements are high (e.g. 256 MB), BCS converges to backfilled space-sharing (BF). Intuitively, when the memory constraints do not allow job multitasking, the system converges to space sharing. This is not true for GS as it experiences sharp degradation in response-time performance, as shown if Figure 6 d) with 128 and 256 MB.

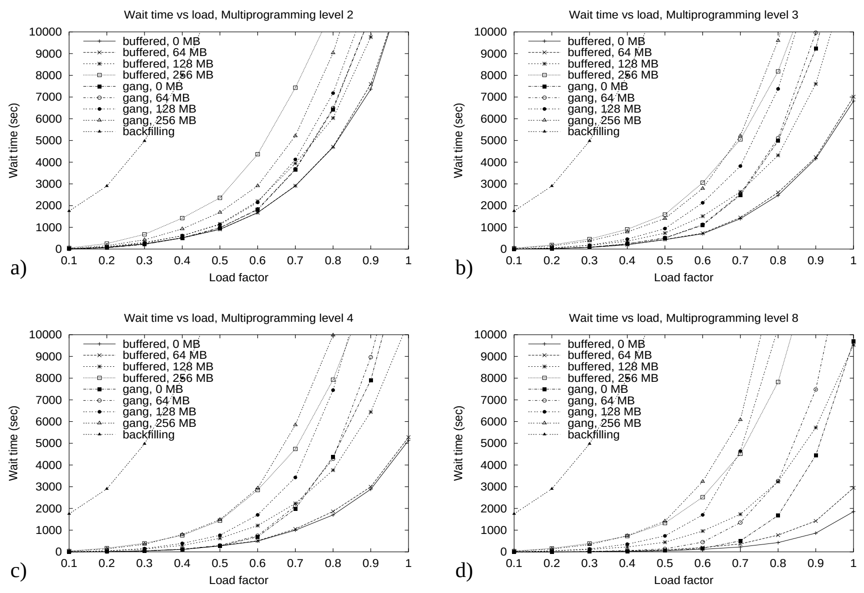

Fig. 7. Wait Time versus Load for Various MPLs.

Figure 7 provides insight on the wait time for the same set of experiments of Figure 6. The graphs clearly show how time sharing can dramatically reduce the wait time over space sharing. BCS reduces the wait time further over backfilled gang scheduling (BGS), in particular with high MPLs.

The reduction of the wait time obtained increasing the MPL, usually implies an increase of the job execution time. In the worst case, the slowdown can be as high as the MPL. In Figure 8 we can see that BCS limits the slowdown when we increase the MPL and outperforms BGS in all configurations, again thanks to the re-use of empty slots in the scheduling matrix at the process level rather than the job level.

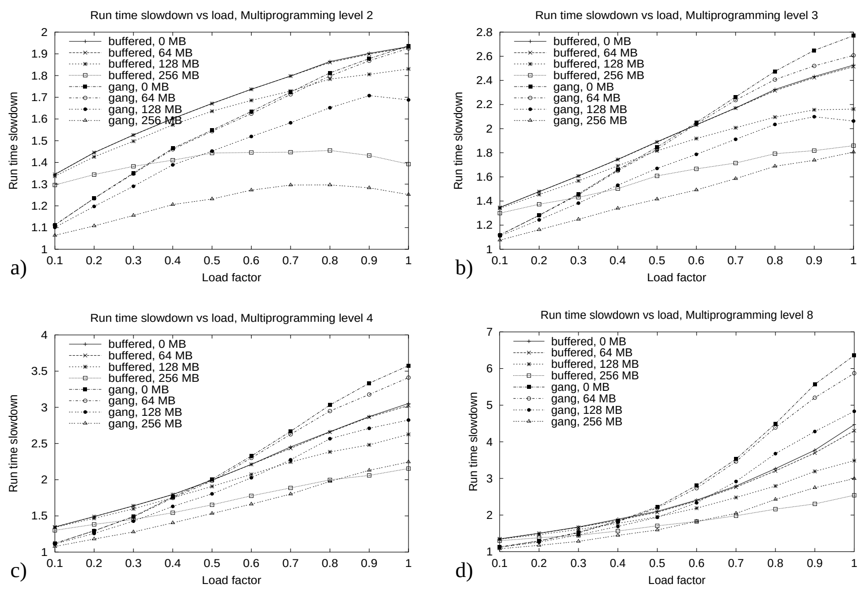

Fig. 8. Execution Time Slowdown versus Load for Various MPLs.

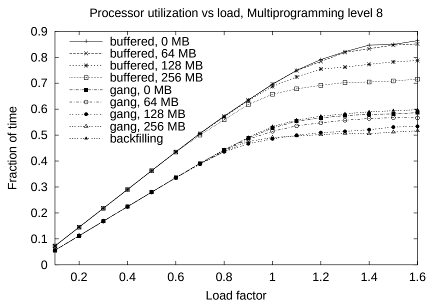

Fig. 9. Processor Utilization versus Load for BCS and BGS.

5.3 Processor Utilization

Most results on job scheduling strategies focus on system utilization rather than processor utilization and show that both BF and BGS can get more than 90% under many workloads. Figure 9 extends these results by analyzing the processor utilization obtained by BF, BGS and BCS.

We observe the following:

- Though BGS improves response time over BF, it does not improve system and processor utilization. The slight decrease in performance is due to the job context-switching overhead.
- With BCS, we can get a processor utilization that asymptotically reach 85%, while BGS and BF approach 60%. This is one of the main advantages of BCS over BGS and BF.
- Processor utilization is sensitive to a job's average memory request when we use time sharing: the higher the memory request, the lower the processor utilization.
- The results address the overlapping of computation and communication only. We expect that the resource utilization gap between BF, BGS and BCS will increase further in the presence of I/O.

6 Discussion

The potential technical impact of BCS is significant for a large class of parallel machines and distributed systems, ranging from Linux clusters to the larger and more sophisticated massively parallel machines. To the best of our knowledge, this is the first methodological attempt to globally optimize the resources of a parallel machine rather than using the limited local knowledge available on each processor.

While BCS enhances overall system performance, particularly with respect to processor utilization and response time, BCS also naturally provides system-level and user-level advantages which we discuss in this section.

6.1 System-Level Advantages

First, the communication is optimized in several ways. The cost of the system calls necessary to access the kernel data structures is amortized over a set of user calls. This implies that the methodology is tolerant to the potential high latencies that can be introduced in a kernel call. BCS can obtain comparable performance to user-level network interfaces (e.g., FM [16] or ST [22]) without using specialized hardware.

Second, 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 limit 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 kernel level to provide fault tolerance in the communication. For example the knowledge of the number of incoming packets greatly simplifies the implementation of receiver-initiated recovery protocols. By globally scheduling a communication pattern, it is also possible to obtain an accurate estimate of the communication time with simple analytical models. By knowing the maximum amount of information that can be delivered in a time-slice, it is possible to minimize the size of the communication buffers in each network interface. This is a crucial problem in a massively parallel architecture. Let's consider, for example, a machine with 10000 processors - the approximate number of processors expected to be in the next ASCI supercomputers. Given that each processor can potentially receive a message from all the remaining 9999 processors, it must reserve a proportional amount of network interface memory (typically few MB for each potential partner). This is infeasible with current network technology and poses a serious limit to the efficient implementation of large scale parallel machines.

Third, because communication is buffered and delayed to the beginning of the next time-slice, we can always implement zero- (or low-, if we desire fault tolerant communication) copy communication. Fault tolerance in general can also be enhanced by exploiting the synchronization point at the end of each time slice to incrementally take a snapshot of the status of the machine.

Fourth, an important advantage of time-sharing parallel jobs is a better utilization of the resources. When we consider I/O, there can be several orders of magnitude of difference between the computational grain of the parallel application and the access time of secondary storage. The usual approach of overlapping computation with I/O, for example using user-level threads, can only provide a limited return in the presence of a single parallel job. By overlapping the activities of multiple parallel jobs we can potentially hide most of the latency. The same argument can be applied to hide the non-uniform latencies of large clusters of SMPs. The higher latency of the inter-cluster communication can be overlapped with the execution of another parallel job.

Fifth, by time-sharing parallel jobs it is possible to obtain better response time and quality of service for critical applications. Time-slicing can be used to give good average completion times for dynamically changing workloads, while retaining fast response times for interactive jobs.

Sixth, because of the deep pipelines and wide out-of-order superscalar architectures of contemporary processors, an interrupt may need to nullify a large number of in-flight instructions [15]. Larger register files require existing system software to save and restore a substantial amount of process state. The reduction of the interrupt frequency provided by BCS can substantially improve the performance on these processors.

Seventh, BCS can also efficiently support future processor architectures, such as Simultaneous Multi-threading (SMT) [3] [5], that time-share multiple processes at hardware level.

6.2 User-Level Advantages

The typical approach to developing parallel software is by using low-level programming models such as MPI. At that level the user is exposed to a large number of details. The user must identify the form of parallelism in the application and decompose it in a set of parallel threads, partition the data set among these threads, map the threads and the data set on a parallel architecture, define communication and synchronization between these threads. This development process is typically specific to a particular application or class of user applications.

As a consequence, *it is extremely difficult and very expensive to build software using such programming models.* Because both correctness and performance can only be achieved by attention to many details, writing optimized MPI programs is a lengthy process, and the result is often machine-dependent[1].

The alternative of using high level programming models, for example automatic parallelization of legacy Fortran codes, is not mature yet and must trade generality in the parallelization process with efficiency, making conservative choices. *BCS has the potential of solving this tradeoff between high development costs and high efficiency vs. low development cost and low efficiency by tolerating several types of inefficiencies related to the parallelization process.*

In a buffered coscheduled system, time-slicing a collection of bad programs (i.e., unbalanced computation or communication) may give the same behavior as a single well-behaved program. Therefore, programs running on a parallel machine need not be carefully balanced by the user to achieve good performance. Multiprogramming can provide opportunities for filling in the "spare communication, computation and I/O cycles" when user programs are sparse, by merging, for example, many sparse communication patterns together to produce a denser communication pattern.

This can have a huge impact on the parallelization of existing legacy codes. If successful, the implementation of BCS could provide a dramatic reduction in the development times and costs of parallel software. Also, *the proposed methodology is valid in general, and not specific to any particular class of applications* (e.g., molecular dynamics, linear solvers, simulations etc.), *nor to a particular machine architecture* (e.g., Cray T3E, SGI, IBM SP).

Finally BCS greatly simplifies the performance evaluation of a parallel application. With BCS the amount of work done by all processors, a metric very close to the sequential complexity of an algorithm, becomes as important as the critical path of the computation.

7 BCS vs BSP

One of the goals of BCS is to transform a collection of unstructured parallel jobs in a single, well-behaved Bulk-Synchronous Parallel (BSP) computation [26] [23].

[1] Though portable to other machines, MPI programs need to go through a non trivial re-optimization process, when moved from one parallel machine to another.

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, BCS 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[2] can be routed with predictable time [10] [21]. 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

$$h_{max} = \frac{T}{g}.$$

Furthermore, by globally scheduling a communication pattern, as described in Section 2, we can derive an accurate estimate of the communication time with simple analytical models already developed for the BSP model [21] [2] [20].

Unfortunately, BSP computations are overly restrictive, and many important applications cannot be efficiently expressed using this model. *With BCS, we can inherit the nice mathematical framework of BSP, without forcing the user to write BSP programs.*

8 Conclusion and Future Work

In this paper, we presented buffered coscheduling (BCS), a new methodology for multitasking jobs in parallel and distributed systems. By leveraging the positive aspects of gang scheduling and dynamic coscheduling, this methodology can significantly improve resource utilization as well as reduce response and wait times of parallel jobs.

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

Using our Job Scheduling Simulator in the presence of memory constraints, we illustrated that backfilling in combination with space sharing or time sharing improves overall system performance. Furthermore, we showed that BCS generally outperformed backfilled gang scheduling and backfilled space sharing.

We also examined how BCS performed with respect to three parameters: type of job communication and synchronization, memory constraints, and processor utilization. We were pleasantly surprised to find that the performance of BCS was relatively insensitive to the communication pattern when the communication was non-blocking communication or, more generally, a collective-communication pattern. In addition, what we originally thought to be a weakness in BCS [6], i.e., memory constraints imposed by BCS, only results in the performance of BCS degrading to being comparable to BF and not significantly worse as with BGS. Finally, the processor utilization with BCS exceeds backfilling gang scheduling (BGS) and BF by as much as 40%.

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. 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.
3. Keith Diefendorff. Compaq Chooses SMT for Alpha: Simultaneous Multithreading Exploits Instruction- and Thread-Level Parallelism. *Microprocessor Report*, 13(16), December 1999.
4. 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.
5. Susan J. Eggers, Henry M. Levy, and Jack L. Lo. Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5), September/October 1997.
6. Fabrizio Petrini and Wu-chun Feng. Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, Cancun, MX, May 2000.
7. Dror G. Feitelson and Morris A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In Dror G. Feitelson and Larry 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 Dror G. Feitelson and Larry 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 Dror G. Feitelson and Larry 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. Joefon Jann, Pratap Pattnaik, Hubertus Franke, Fang Wang, Joseph Skovira, and Joseph Riordan. Modeling of Workload in MPPs. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 95–116. Springer-Verlag, 1997.
14. 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.
15. Stephen W. Keckler, Andrew Chang, Whay S. Lee, Sandeep Chatterje, and William J. Dally. Concurrent Event Handling through Multithreading. *IEEE Transactions on Computers*, 48(9):903–916, September 1999.
16. Mario Lauria and Andrew Chien. High-Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, November 1995.
17. Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph. Implications of I/O for Gang Scheduled Workloads. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
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. William E. Weihl Patrick Sobalvarro, Scott Pakin and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 231–256. Springer-Verlag, 1998.
20. 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.
21. 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.
22. Ian 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.
23. D. B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Journal of Scientific Programming*, 1998.
24. Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The EASY-LoadLeveler API Project. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer-Verlag, 1996.

25. Patrick Sobalvarro and William 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.
26. Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.
27. Yanyong Zhang, Hubertus Franke, José Moreira, and Anand Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, Cancun, MX, May 2000.