Proceedings of the 7th Int'l Conference on Advanced Computing and Communications (ADCOM'99), Dec. 1999. LA-UR 01-3025

Efficient Scheduling of Parallel Jobs on Massively Parallel Systems*

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

† Computing, Information, & Communications
Los Alamos National Laboratory
Los Alamos, NM 87545

§ School of Elec. & Comp. Eng.
Purdue University
W. Lafayette, IN 47907

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. 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 has long been an active area of research [6, 7]. 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 [11].

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 [5] 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 [12].

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 [8].

An intermediate approach initially developed at UC Berkeley and MIT in recent years is implicit or dynamic coscheduling [1, 4, 14]. 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 [13].

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 these limitations are successfully addressed in [13] 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 (a technique similar to Periodic Boost), strobing, and non-blocking, one-sided communication. By leveraging these

*This work was supported by the U.S. Dept. of Energy through Los Alamos National Laboratory contract W-7405-ENG-36.

Proceedings of the 7th Int’l Conference on Advanced Computing
and Communications (ADCOM’99), Dec. 1999. LA-UR 01-3025

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 (*a la* CISC$\rightarrow$RISC instruction-set simplification).

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 *global* processor and network utilization (i.e., the number of active processors and the fraction of active links) during the execution of a transpose FFT algorithm on a parallel machine with 256 processors. These processors are connected with an indirect interconnection network using state-of-the-art routers. Based on these figures, there is obviously an *uneven and inefficient use of system resources.* During the two computational phases of the transpose, the network is idle. Conversely, when the network is actively transmitting messages, the processors are not doing any useful work.

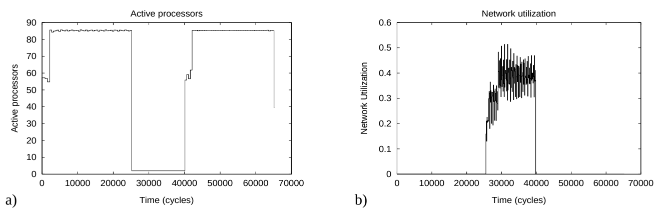

Figure 1: Resource Utilization in a Transpose FFT.

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 [15].

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] (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) [2] without using specialized hardware.

3.2 Strobing

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

In order to provide the above capability, we propose a strobing mechanism to support the scheduling of a set of parallel jobs which share a parallel machine. Let us assume that each parallel job runs on the entire set of p processors, i.e., jobs are time-sharing the whole machine. 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, the user process regains control of the processor; and at the end of it, 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.

Proceedings of the 7th Int'l Conference on Advanced Computing and Communications (ADCOM'99), Dec. 1999. LA-UR 01-3025

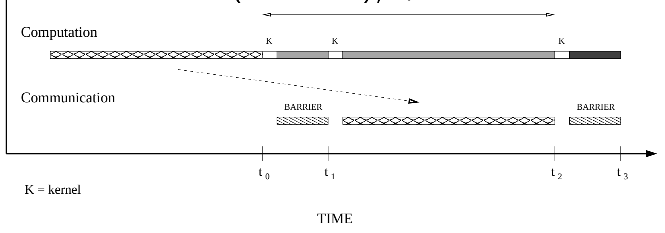

Figure 2: Scheduling Computation and Communication. Communication accumulated in the time-slice up to t_0 is downloaded into the network between t_1 and t_2 (after the barrier synchronization). $\delta \equiv$ length of a time-slice $= t_2 - t_0$.

This strategy can be easily extended to deal with space-sharing where different regions run different sets of programs [10]. 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.

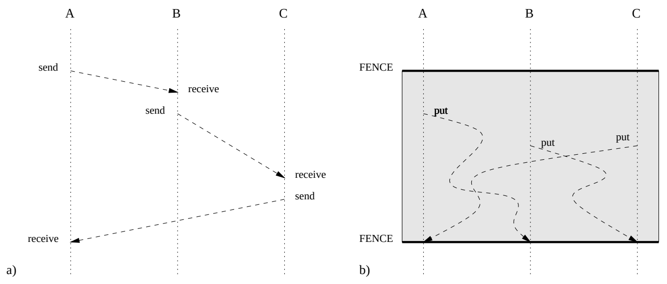

Figure 3: (a) Message Passing (b) 1-Sided Communication.

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.

4 Experimental Results

Our preliminary experimental results include a working implementation of a representative subset of MPI-2 on a detailed (register-level) simulation model [16]. The simulation environment includes a standard version of MPI-2 and a multitasking version which 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 and vary the computational granularity and load imbalance. As in [4], our workloads consist of a collection of single-program multiple-data (SPMD) parallel jobs that alternate phases of purely local computation with phases of interprocess communication. A parallel job generated by one such program consists of a group of P processes where 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 variance for a particular computational granularity g, and thus, the degree of load imbalance across processors.

The communication phase of our workloads consists of an opening barrier, followed by an optional sequence of pairwise communication events separated by small amounts of local computation, c, and finally an optional closing barrier. We consider two communication patterns: *Barrier* and *Transpose*. *Barrier* consists of only the opening barrier and thus contains no additional dependencies. This workload can be used to analyze how buffered coscheduling responds to load imbalance. *Transpose* is a communication-intensive workload. It tries to emulate the communication pattern generated by the FFT transpose algorithm [9], where each process accesses data on all other processes.

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

We assume a processor speed of 500 MHz and a network of 32 processors interconnected in a 5-dimensional cube topology with performance characteristics similar to those of Myrinet routing and network cards [3]. This network features a one-way data rate of about 1 Gbit/s and a base network latency of few μs. The simulator models the following at register level: the congestion inside the network and at the network interface and the routing and flow-control protocols.

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

Proceedings of the 7th Int'l Conference on Advanced Computing
and Communications (ADCOM'99), Dec. 1999. LA-UR 01-3025

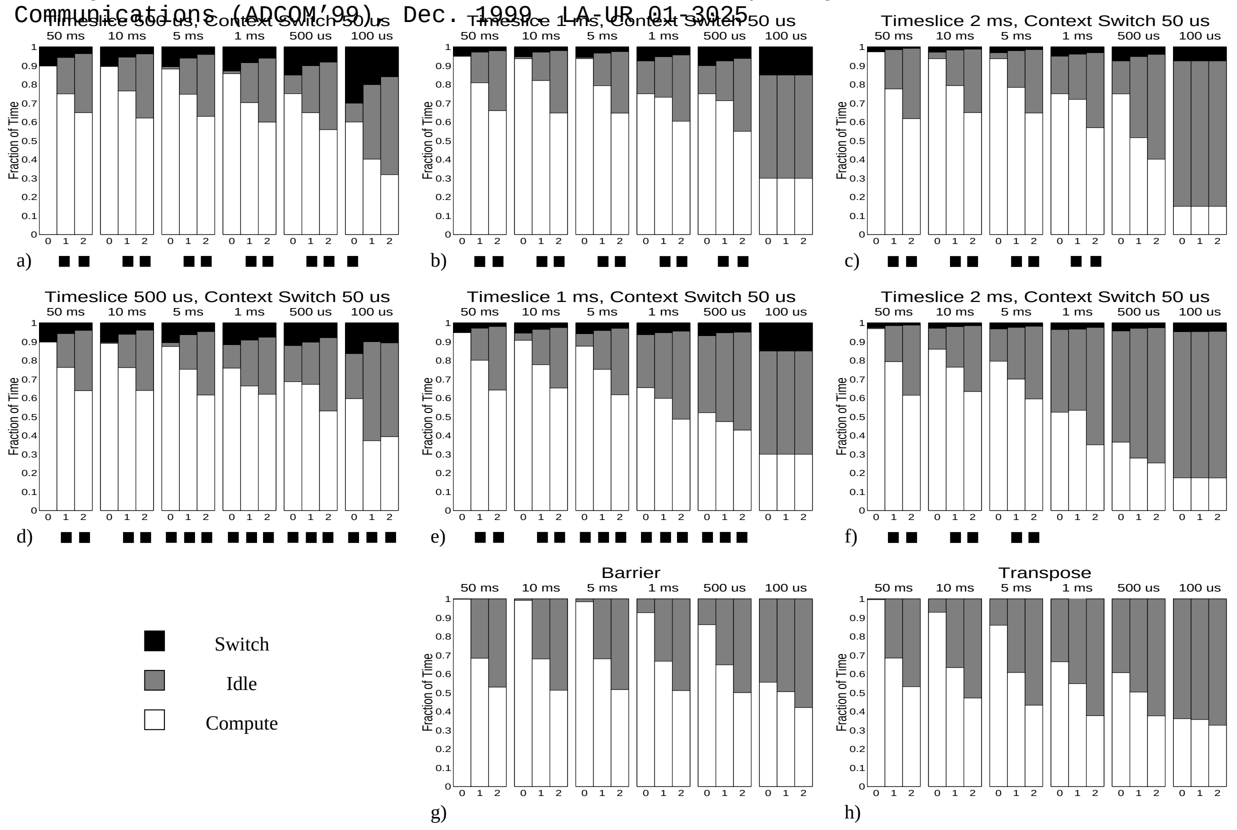

Figure 4: Execution characteristics as a function of computational granularity, load imbalance and context-switch latency. A black square under a bar highlights the configurations where buffered coscheduling gets better resource utilization.

4.1 Resource Utilization and Throughput

Figures 4(a)-(c) and 4(d)-(f) illustrate the communication and computation characteristics of the *Barrier* and *Transpose* benchmarks, respectively, as a function of the computational granularity, load imbalance and context-switch penalty. Each bar shows the fraction of time spent in one of the following states, averaged over all processors: computing, context-switching and idling.

For each of the benchmarks, we consider time-slices of 500 μs, 1 ms and 2 ms. And for each of the time-slice alternatives, the computational granularity varies from a coarser-grained 50 ms to a finer-grained 100 μs. Finally, for each of the computational granularities, the load imbalance is non-existent, i.e., $v = 0$ (no variance), $v = 1 * g$ (the variance is equal to the computational granularity), and $v = 2 * g$ (the variance is equal to twice the computational granularity, i.e., high degree of load imbalance).

Figures 4(g) and (h) show the breakdown for the *Barrier* and *Transpose* workloads when they are run in dedicated mode on the standard MPI-2 run-time support (i.e., a single job is run until completion without multitasking).

For Figures 4(a)-(f), a black square under a bar highlights the configurations where multitasking produces better resource utilization than batch scheduling. For instance, running a *Transpose* job in batch mode with a computational granularity of 100 μs and no load imbalance (i.e., Figure 4(h), the "0" bar in the rightmost group of bars) produces 35% processor utilization. When we run the same job but multitasked with two other jobs of the same kind, the utilization can be as high as 60% (see Figure 4(d)), a 71% improvement in utilization.

The results reported in [13] 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 70% (TRANSPOSE with a computational grain size of 100 μs). Our experimental results show that in a large majority of cases, we experience *speedup* rather than slowdown. And even in the worst case, the slowdown is less than 10%. Though the results are not directly comparable, the performance difference between our scheduling methodology and those reported in [13] is significant enough to lead us to believe that buffered coscheduling can bridge the gap between the "faster response time but lower throughput" of existing coscheduling algorithms and the "slower response time but higher throughput" of batch scheduling.

4.2 Sensitivity Analysis

By examining the breakdowns of each bar, we can see several important trends. First, 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.

Second, the time-slice length is a critical parameter in

Proceedings of the 7th Int'l Conference on Advanced Computing and Communications (ADCOM'99), Dec. 1999, LA-UR 01-3025

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.

Third, in Figures 4(a) and (d), we see that using a relatively small time-slice in our multitasking environment produces higher processor utilization than when a single job runs in a dedicated environment in 11 out of 18 cases for *Barrier* and 16 out of 18 for *Transpose*. 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.

Fourth, the experimental results show that overall performance is sensitive to time-slice length. For example, for the time-slices we chose for our experiments, the smallest time-slice (500 μs) produced the best overall performance. Thus, this implies that selecting the "correct" time-slice length is very important.

As a rule of thumb, multitasking leads to good performance as long as the average computational grain size is larger than the time-slice and the time-slice, on its turn, is sufficiently larger than the context-switch penalty.

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.

5 Conclusion and Future Work

In this paper, we presented buffered coscheduling, a new methodology to multitask parallel jobs on a parallel computer in order to achieve efficient resource utilization and improved system throughput. Buffered coscheduling addresses the main limitation of explicit coscheduling — the high latency needed to perform a global context switch. It also provides a simple framework to increase resource utilization, simplify the design of the run-time support, enhance fault tolerance, and perform effective global optimizations.

While batch scheduling results in better system throughput, batching is particularly unattractive for interactive jobs due to poor response times. On the other hand, existing coscheduling algorithms produce better response times but result in slowdown, and hence, reduced system throughput. In contrast, buffered coscheduling not only provides better response times but also increases system throughput.

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

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

[3] 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.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.