A New Approach to Parallel Program Development and Scheduling of Parallel Jobs on Distributed Systems

Fabrizio Petrini and Federico Bassetti *
Scientific Computing Group (CIC-19)
Los Alamos National Laboratory

Alexandros Gerbessiotis
Department of Computer and Information Science
New Jersey Institute of Technology

Abstract *A typical way to increase the performance of a parallel program on a given parallel platform is to try to overlap computation and communication in order to decrease running time and simultaneously increase efficiency (speedup). This approach, however, leads to diminishing returns after an initial and time-consuming development phase. We propose an alternative approach to program development and resource utilization in a parallel machine. Rather than increasing the complexity of a single program, we attempt to* overlap *the execution of two or more parallel jobs on the same machine. Though this approach is not new, we think that the adoption of one-sided communication in program development and global scheduling strategies can pave the way to a new methodology of developing parallel programs that are simpler and can use more efficiently a wide range of parallel machines and distributed systems. This methodology is based on 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.*

Keywords: parallel job scheduling, communication libraries, run-time support, operating systems.

1 Introduction

The scheduling of parallel jobs has long been an active area of research [3]. It is a challenging problem because the performance and applicability of parallel scheduling algorithms is highly dependent upon factors at different levels: the workload, the parallel programming language, the operating system (OS), and the machine architecture.

Time-sharing scheduling algorithms are particularly attractive because they can provide good response time without migration or predictions on the execution time of the parallel jobs. However, 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 [8].

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 [2] 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, 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

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

parallel jobs interacts poorly with interactive jobs and jobs performing I/O [9].

At the other end of the spectrum is local scheduling, 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 explicit coscheduling because the scheduling is not coordinated across processors [5].

An intermediate approach developed at UC Berkeley and MIT in recent years is implicit or dynamic coscheduling [1, 15]. With implicit coscheduling, each local scheduler makes independent decisions that dynamically coordinate the scheduling actions of cooperating processes across processors. These actions are based on local events that occur naturally within communicating applications. For example, on message arrival, a processor speculatively assumes that the sender is active and will 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].

The main drawbacks of implicit coscheduling include (1) the potential inefficiency with jobs displaying fine-grained communication, (2) the limited programming model supported, (3) the limitation of a localized flow-control strategy, (4) the non-trivial implementation of fault tolerance, and (5) the lack of a reliable performance model of the execution time of parallel jobs, due to the dynamic interleaving of several jobs. We elaborate on a few of these drawbacks below.

To address the above limitations, we present a new methodology to schedule parallel jobs. This methodology conjugates the positive aspects of explicit and implicit coscheduling using three innovative techniques: communication buffering to amortize communication overhead; strobing to globally exchange information at regular intervals; and non-blocking, one-sided communication to decouple communication and synchronization. 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 the proposed methodology 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 (*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 scientific applications, e.g., resource utilization and communication access patterns, and which inspired our proposed methodology. The methodology itself is described in Section 3, some preliminary results are presented in Section 4 and some concluding remarks are given 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 [13]. 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. These characteristics are shared by many SPMD programs, including Accelerated Strategic Computing Initiative (ASCI) application codes such as Sweep3D [6]. Hence, there is tremendous potential for increasing resource utilization in a parallel machine.

Another important characteristic shared by many scientific parallel programs is their access pattern to the network. The vast majority of scientific applications display *bursty communication patterns* with alternating spikes of impulsive communication with periods of inactivity.

In Figure 2, we show the network utilization by running four distinct scientific applications over a parallel machine with 256 processors [10]. In all four cases, we can identify *communication holes*, i.e., periods of network inactivity, in the network. Therefore, there exists a significant amount of communication bandwidth which can be made available for other purposes.

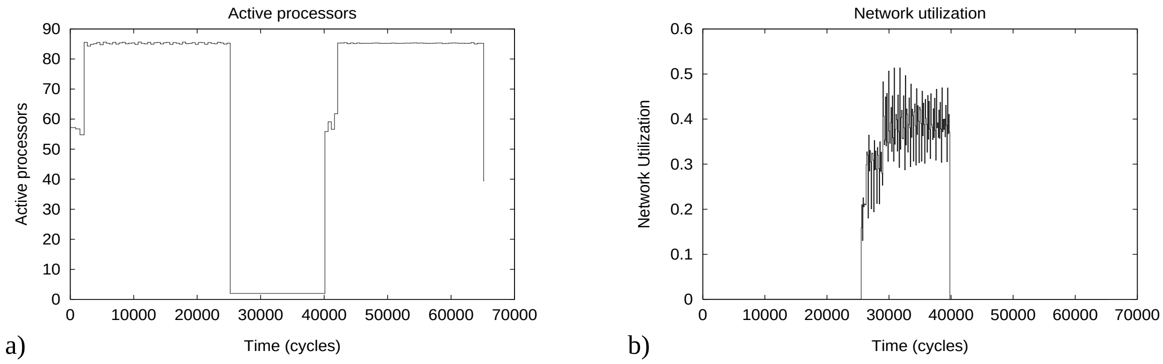

Figure 1: Resource Utilization in a Transpose FFT Algorithm.

3 Multitasking Parallel Jobs

In order 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.*

We propose three techniques to implement the multitasking of parallel jobs. (1) 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 the communication, we allow for the global scheduling of the communication pattern. (2) A strobing mechanism performs a total exchange of information at the end of each time-slice so that multiprocessor machines may move away from isolated scheduling algorithms [1, 15] (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. (3) Finally, we propose to use non-blocking and one-sided communication primitives to decouple communication and synchronization in order to schedule the communication pattern 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 our methodology can be tolerant to the potentially high latencies that can be introduced in a kernel call or in the initialization of the NIC that can reside on a slow I/O bus. In addition to amortizing communication and scheduling overhead, we 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) [7].

3.2 Strobing

The uneven resource utilization and the periodic, bursty communication patterns generated by many parallel scientific applications can be addressed by performing a total exchange of information and synchronizing the 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

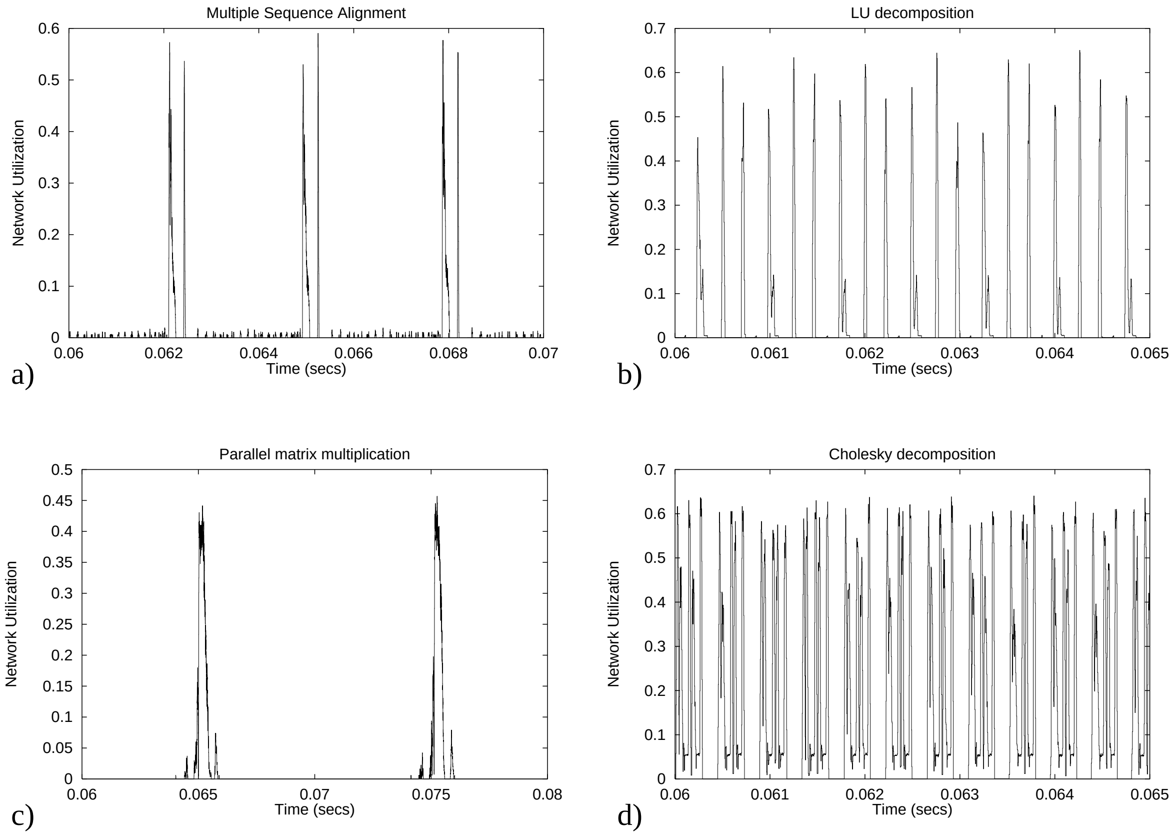

Figure 2: Network Utilization in Scientific Parallel Programs.

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 and also triggers the downloading of any buffered packets into the network. The strobe is implemented by designating one of the processors as the *master*, the one who generates the "heartbeat" of the strobe. The generation of heartbeats will be achieved by using a timeout mechanism which can be associated with the network interface card (NIC). This ensures that strobing incurs very little CPU overhead.

On reception of the heartbeat, each processor (excluding the master), is interrupted and downloads a broadcast heartbeat into network. After downloading the heartbeat, the processor continues running the currently active job. (This ensures computation is overlapped with communication.) When p heartbeats arrive at a processor, the processor will enter a strobing 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.

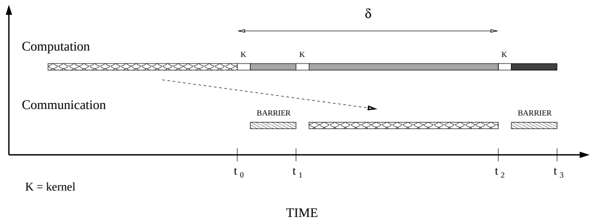

Figure 3: Scheduling Computation and Communication. The communication accumulated before t_0 is downloaded into the network between t_1 and t_2 (after the completion of the barrier synchronization).

Figure 3 outlines how computation and communication can be scheduled over a generic processor. At the beginning of the heartbeat, t_0, the control is given to the kernel, which downloads the control packets for the 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 before t_0 to be delivered in the next time-slice. At t_1, the processor will know the number of incoming packets that it is going to receive in the next communication time-slice as well as the sources of the packets and will start the downloading of outgoing packets.

This strategy can be easily extended to deal with space-sharing where different regions run different sets of programs [2]. In this case too, all these regions are synchronized by the same heartbeat.

The total-exchange can be properly optimized by exploiting the low-level features of the interconnection network. For example, if control packets are given higher priority than the standard background traffic at the sending and receiving endpoints, they can be delivered with predictable latency.

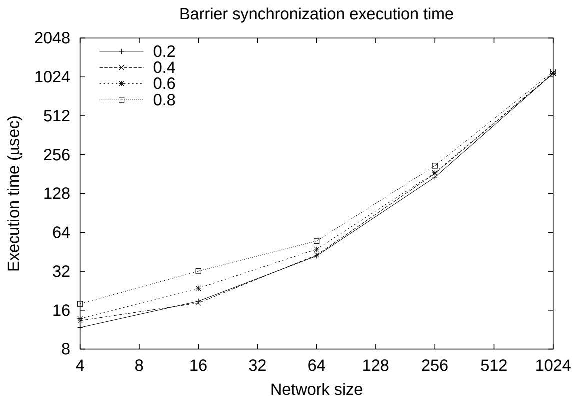

Figure 4: Execution time of the total exchange algorithm in a family of interconnection networks with up to 1024 processing nodes.

We analyzed the execution time of the direct total-exchange algorithm in a family of indirect networks with up to 1024 processing nodes. In this experiment, whose results are shown in Figure 4, we assume the existence of background traffic that varies from 20% to 80% of the network capacity. We can see that the execution time is largely insensitive to the intensity of the background traffic. With 64 processing nodes (the configuration of a single SGI Origin 2000 cluster) the execution time is only 50 μsec and this increases to 150 μsec with 256 nodes. Due to the quadratic increase of the number of messages sent during the total-exchange, the execution time reaches 1 msec with 1024 nodes, limiting the scalability of the approach. This problem can be addressed in a clustered architecture, like ASCI Blue Mountain, by using a multi-phase, indirect algorithm, that in the first phase executes the total-exchange inside each single cluster, then performs a total-exchange between clusters, to conclude with a final phase internal to the clusters, giving a barrier synchronization time of less than 300 μsec.

Using these basic results, we can eliminate the transmission of the ghost packets, those that do not contain any useful information and are sent only for synchronization purposes, by adopting the following strategy: if a control packet is not received within the communication threshold, which is determined by observing the maximum latency experienced by the control packets as shown in Figure 4, we can safely assume that there is no pending communication in the source processor and that no packets will be sent in the next time-slice.

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

3.3 Blocking vs. Non-Blocking

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

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

Let us consider the following example. The dynamics of a message-passing program can be represented as a two-dimensional graph with processes on the horizontal axis and time on the vertical one, as shown in Figure 5. Arrows between processes represent communication between a sender and a receiver. In Figure 5(a), three processes

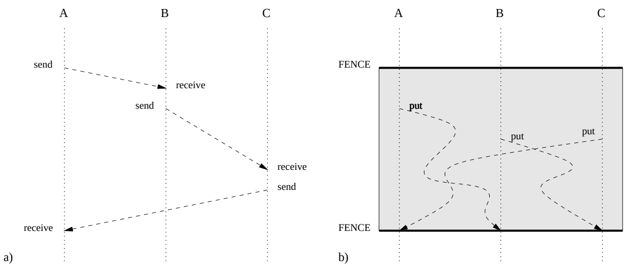

Figure 5: (a) Message Passing. (b) One-Sided Communication.

exchange messages. For the sake of convenience, let us assume that there is no dependency between the messages (i.e., they can be sent in any order). Using a traditional, blocking, message-passing programming style, we must define a communication schedule even if one is *not* required, e.g., A sends to B, B receives from A and sends to C, C receives from B and sends to A.

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

3.4 Bulk-Synchronous Parallel Programs

Using our proposed strobing and buffering mechanisms, any generic parallel program can be transformed into a Bulk-Synchronous Parallel (BSP) one [14]. 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 execution times of parallel programs.

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 [4, 11]. 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.

4 Preliminary Results

Our preliminary results include a working implementation of a subset of MPI-2, which includes *all* the communication primitives used by the ASCI applications plus basic primitives to implement one-sided communication. This implementation is used to test our presented methodology.

The experimental results obtained by running two or more instances of the ASCI code Sweep3D show that is possible to overlap virtually all the computation of one parallel job with the communication of another parallel job. We have also seen that by using time-slices of 100 msec (or larger) it is possible to hide all the overhead generated by the strobing and scheduling algorithms. The experimental evaluation uses a detailed (register-

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

level) simulation model [12] and will be shortly followed by an implementation on a Linux cluster.

5 Conclusion

In this paper, we have present 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. This methodology is based on 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 advantages of the proposed methodology 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.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Highly Efficient Gang Scheduling Implementation. In *Supercomputing 98*, Orlando, FL, November 1998.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] Fabrizio Petrini and Marco Vanneschi. Latency and Bandwidth Requirements of Massively Parallel Programs: FFT as a Case Study. *Future Generation Computer Systems*, 1999. Accepted for publication.

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

[15] 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.