LA-UR- 00-894

Approved for public release; distribution is unlimited.

Title: Scheduling with Global Information in Distributed Systems

Author(s): Fabrizio Petrini and Wu-chun Feng

Submitted to: The 20th International Conference on Distributed Computing Systems

Scheduling with Global Information in Distributed Systems *

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

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

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

Abstract

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

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

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

1. Introduction

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

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

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

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

At the other end of the spectrum is local scheduling, where each processor independently schedules its processes. While this approach is attractive due to its ease of construction, the performance of fine-grain communicating

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

buffering can theoretically achieve performance comparable to user-level network interfaces (i.e., OS-bypass protocols) [2] without using specialized hardware.

3.2 Strobing

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

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

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

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

Figure 2 outlines how computation and communication can be scheduled over a generic processor. At the beginning of the heartbeat, t_0, the kernel downloads control packets for the total exchange of information. During the execution of the barrier synchronization, the user process then regains control of the processor; and at the end of it, the kernel schedules the pending communication accumulated before t_0 to be delivered in the current time slice, i.e., δ. At t_1, the processor will know the number of incoming packets that it is going to receive in the communication time-slice as well

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

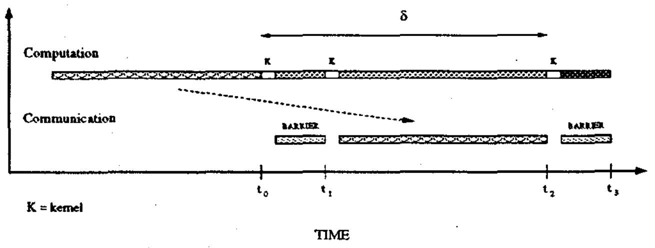

Figure 2. Scheduling Computation and Communication. The communication accumulated before t_0 is downloaded into the network between t_1 and t_2.

as the sources of the packets and will start the downloading of outgoing packets.

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

The total exchange of information can be properly optimized by exploiting the low-level features of the interconnection network. For example, if control packets are given higher priority than background traffic at the sending and receiving endpoints, they can be delivered with predictable network latency[2] during the execution of a direct total-exchange algorithm[3] (Figure 3). We generated

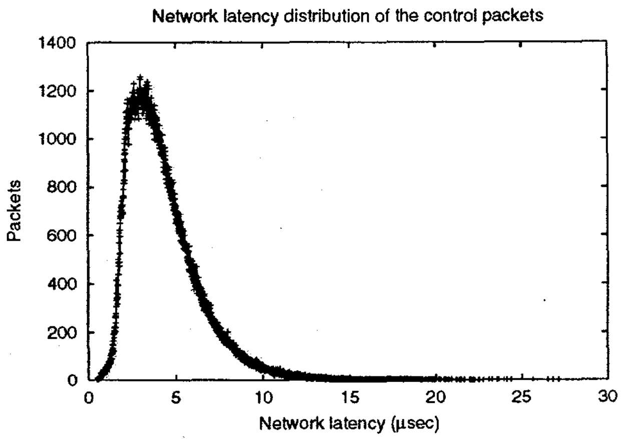

Figure 3. Network Latency Distribution.

this distribution using a network of 256 processing nodes equipped with wormhole routers similar to those in the SGI

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

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

communicating processes do *not* need to be simultaneously scheduled to perform the communication.

3.4 Bulk-Synchronous Parallel Programs

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

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

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

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

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

4 Experimental Results

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

4.1 Characteristics of the Synthetic Workloads

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

We consider two communication patterns: *Barrier* and *Transpose*. *Barrier* consists of only the opening barrier and thus contains no additional dependencies. This workload can be used to analyze how our methodology responds to load imbalance. *Transpose* is a communication-intensive workload that emulates the communication pattern generated by the FFT transpose algorithm

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

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

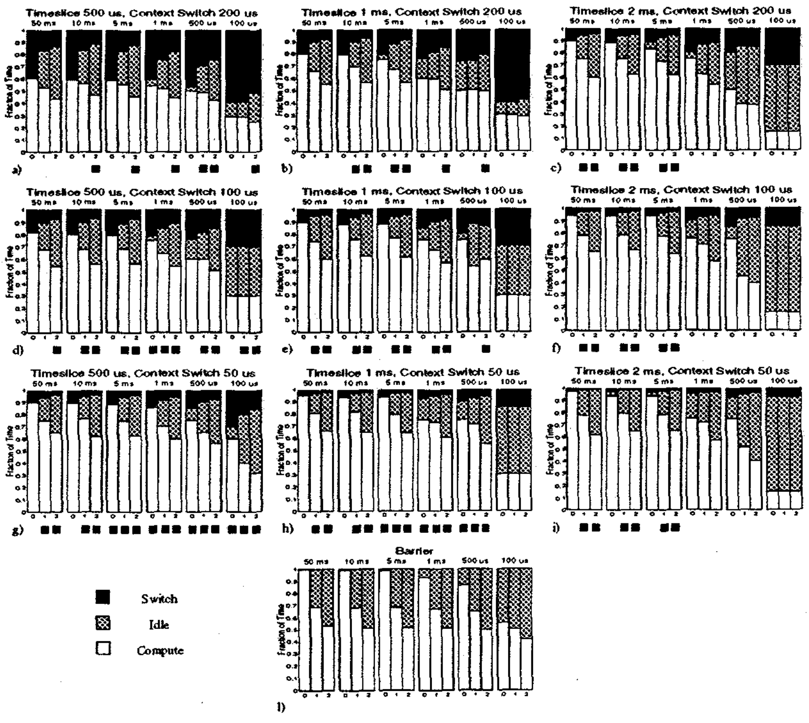

Figure 6. Execution Characteristics of the *Barrier* Workload.

5. Conclusion

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

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

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

References

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

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

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

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

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

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

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

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