Proceedings of the 2000 International Parallel and Distributed Processing Symposium (IPDPS 2000).

Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on 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 scheduling methodology for time-sharing communicating processes in parallel and distributed systems. The methodology has two primary features: communication buffering and strobing. With communication buffering, communication generated by each processor is buffered and performed at the end of regular intervals to amortize communication and scheduling overhead. This infrastructure is then leveraged by a strobing mechanism to perform a total exchange of information at the end of each interval, thus providing global information to more efficiently schedule communicating processes.

This paper describes how buffered coscheduling can optimize resource utilization by analyzing workloads with varying computational granularities, load imbalances, and communication patterns. The experimental results, performed using a detailed simulation model, show that buffered coscheduling is very effective on fast SANs such as Myrinet as well as slower switch-based LANs.

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

1. Introduction

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*.

Explicit coscheduling [5] 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 [10].

Conversely, local scheduling allows each processor to independently schedule its processes. Although attractive due to its ease of construction, the performance of fine-grain communicating jobs degrades significantly because scheduling is not coordinated across processors [7].

An intermediate approach developed at UC Berkeley and MIT is implicit or dynamic coscheduling [1, 4, 12, 16] where each local scheduler makes 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 likely send more messages in the near future.

In this paper, we present a new methodology that conjugates the positive aspects of explicit and implicit coscheduling using three techniques: communication buffering to

*This work was supported by the U.S. Dept. of Energy through Los Alamos National Laboratory contract W-7405-ENG-36. This paper is LA-UR 00-892.

amortize communication overhead (a technique similar to *periodic boost* [11]); 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 rest of the paper is organized as follows. Section 2 describes the motivation and features of buffered coscheduling. Preliminary results are presented in Section 3. Finally, we present our conclusions in Section 4.

2 Multitasking Parallel Jobs

Our study of resource utilization in SPMD programs inspired our buffered coscheduling methodology which consists of communication buffering, strobing, and optionally non-blocking communication. This methodology allows all the communication and I/O which arise from a *set* of parallel programs to be overlapped with the computations in those programs.

2.1 Motivation

Figure 1 shows the global processor and network utilization during the execution of an FFT transpose algorithm on a parallel machine with 256 processors connected with an indirect interconnection network using state-of-the-art routers [3]. Based on these figures, we observe an *uneven and inefficient use of system resources*. These characteristics are shared by many SPMD programs, including unclassified ASCI application codes such as Sweep3D [8]. Hence, there is tremendous potential for increasing resource utilization in a parallel machine.

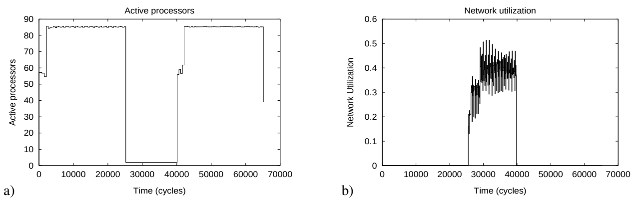

Figure 1. Resource Utilization in an FFT Transpose Algorithm.

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 [13]. Thus, there exists a significant amount of unused network bandwidth which could be used for other purposes.

2.2 Communication Buffering

Instead of incurring communication and scheduling overhead on a per-message basis, we accumulate the communication messages generated by each processor and amortize the overhead over a set of messages. By delaying the communication, we allow for the global scheduling of the communication pattern. And because we can implement zero-copy communication, this technique can theoretically achieve performance comparable to OS-bypass protocols [2] without using specialized hardware.

2.3 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.

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. 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 is 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.

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 all the 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

[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.

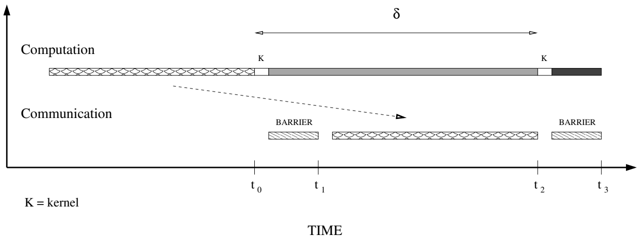

Figure 2. Scheduling Computation and Communication. The communication accumulated before t_0 is downloaded into the network between t_1 and t_2.

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 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 [5, 9, 17]. 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] [14].

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. The same information can be used at the kernel level to provide fault-tolerant communication. For example, the knowledge of the number of incoming packets greatly simplifies the implementation of receiver-initiated recovery protocols.

[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.

3 Experimental Results

As an experimental platform, our working implementation includes a representative subset of MPI-2 on a detailed (register-level) simulation model [15]. The run-time support on this platform includes a standard version of a substantive subset of MPI-2 and a multitasking version of the same subset that implements the main features of our proposed methodology. It is worth noting that the multitasking MPI-2 version is actually much simpler than the sequential one because the buffering of the communication primitives greatly simplifies run-time support.

3.1 Characteristics of the Synthetic Workloads

As in [4], 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 three communication patterns: *Barrier*,*News*, and *Transpose*. *Barrier* consists of only the closing barrier and thus contains no additional dependencies. We can therefore use this workload to analyze how buffered coscheduling responds to load imbalance. The other two patterns consist of a sequence of remote writes. The communication pattern generated by *News* is based on a stencil with a grid, where each process exchanges information with its four neighbors. This workload represents those applications that perform a domain decomposition of the data set and limit their communication pattern to a fixed set of partners. *Transpose* is a communication-intensive workload that emulates the communication pattern generated by the FFT transpose algorithm [6], where each process accesses data on all other processes.

For our synthetic workload, we consider three parallel jobs with the same computational granularities, load imbalances, and communication patterns arriving at the same time in the system. The communication granularity, c, is fixed at 8 μs. The number of communication/computation iterations is scaled so that each job runs for approximately one second in a dedicated environment. The system consists of 32 processors, and each job requires 32 processes (i.e. jobs are only time-shared).

3.2 The Simulation Model

The simulation tool that we use in our experimental evaluation is called SMART (Simulator of Massive ARchitectures and Topologies) [15], a flexible tool designed to model the fundamental characteristics of a massively parallel architecture. The current version of SMART is based on the x86 instruction set. The architectural design of the processing nodes is inspired by the Pentium II family of processors. In particular, it models a two-level cache hierarchy with a write-back L1 policy and non-blocking caches.

Our experiments consider two networks with 32 processing nodes, representative of two different architectural solutions. The first network is a 5-dimensional cube topology with performance characteristics similar to those of Myrinet routing and network cards [3]. This network features a one-way data rate of about 1 Gb/s and a base network latency of less than a μs. The second network is based on a 32-port, 100-Mb/s Intel Express switch, a popular solution due its attractive performance/price ratio.

3.3 Resource Utilization

Figures 3 and 4 show the communication/computation characteristics of our synthetic benchmarks on a Myrinet-based interconnection network and an Intel Express switch-based network, respectively, as a function of the communication pattern, granularity, load imbalance, and time-slice duration. Each bar shows the percentage of time spent in one of the following states (averaged over all processors): computing, context-switching and idling.

For each communication pattern in the Myrinet-based network, we consider time-slices of 0.5, 1, and 2 ms. In contrast, for the switch-based network, we consider time-slices of 2, 4, and 8 ms due to the larger communication overhead and lower bandwidth. In both cases, the context-switch penalty is 25 μs.

In each group of three bar graphs, the computational granularity is the same, but the load imbalance is increased as a function of the granularity itself, i.e., $v = 0$ (i.e. no variance), $v = g$ (the variance is equal to the computational granularity) and $v = 2g$ (high degree of imbalance).

Figures 3 (l)-(n) and 4 (l)-(n) show the breakdown for the *Barrier*, *News*, and *Transpose* workloads when they are run in dedicated mode with standard MPI-2 run-time support. For Figures 3 (a)-(i) and 4 (a)-(i), a black square under a bar denotes a configuration where buffered coscheduling achieves better resource utilization than MPI-2 user-level communication, and a circle indicates a configuration where the performance loss of buffered coscheduling is within 5%.

Based on Figures 3 and 4, we make the following observations. First, the performance of buffered coscheduling is sensitive to the context-switch latency. As context-switch latency decreases, resource utilization and throughput improve. Second, as the load imbalance of a program increases, the idle time increases. Third, and most importantly, these initial results indicate that the time-slice length is a critical parameter in determining overall performance. A short time-slice can achieve excellent 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 it cannot reduce the load imbalance, particularly in the presence of fine-grained computation.

In Figures 3 (a), (d), and (g) which use a relatively small time-slice in a Myrinet-based network, buffered coscheduling produces higher processor utilization than when a single job runs in a dedicated environment in over 55% of the cases and produces higher (or no worse than 5% less) resource utilization in nearly 75% of the cases.

Taking a big picture view of Figure 3, we conclude that for high-performance Myrinet-like networks that buffered coscheduling performs admirably as long as the average computational grain size is larger than the time-slice and the time-slice in turn is sufficiently larger than the context-switch penalty. In addition, when the average computational grain size is larger than the time-slice, the processor utilization is mainly influenced by the degree of imbalance.

With a less powerful interconnection network, we find that buffered coscheduling is even more effective in enhancing resource utilization. Figures 4 (a), (d), and (g) show that in a 100-Mb/s switch-based interconnection network, buffered coscheduling outperforms the basic approach in 16 out of 18 configurations with *Barrier*, 13 out 18 with *News*, and 10 out 18 with *Transpose*. In this last case, the performance of buffered coscheduling can be improved by increasing the time-slice.

What makes buffered coscheduling so much more effective in a less powerful interconnection network? The answer lies in the "excessive" communication overhead that is incurred in these commodity networks when each job is run in dedicated mode with MPI-2 run-time support; the overhead is high enough to adversely impact the resource utilization of the processor and network. For example, by comparing the graphs for the 500-μs computational granularity in Figures 3 (l)-(n) and Figures 4 (l)-(n), respectively, we see that the resource utilization for the switch-based network is significantly lower than the Myrinet network when running in dedicated mode. Consequently, there is substantially more room for resource-utilization improvement in the switch-based network, and the buffered coscheduling methodology takes full advantage this by overlapping computation with potentially long communication delays/overhead, thus hiding the communication overhead.

Irrespective of the type of network, for the cases where jobs are perfectly balanced, i.e., $v = 0$, running a sin-

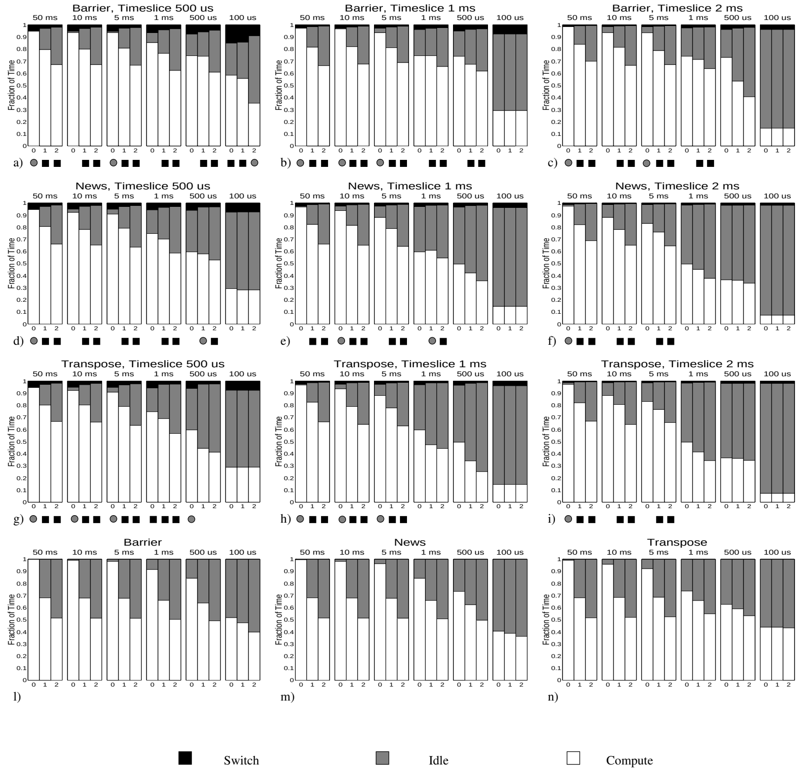

Figure 3. Resource Utilization on a Myrinet-Based Interconnection Network.

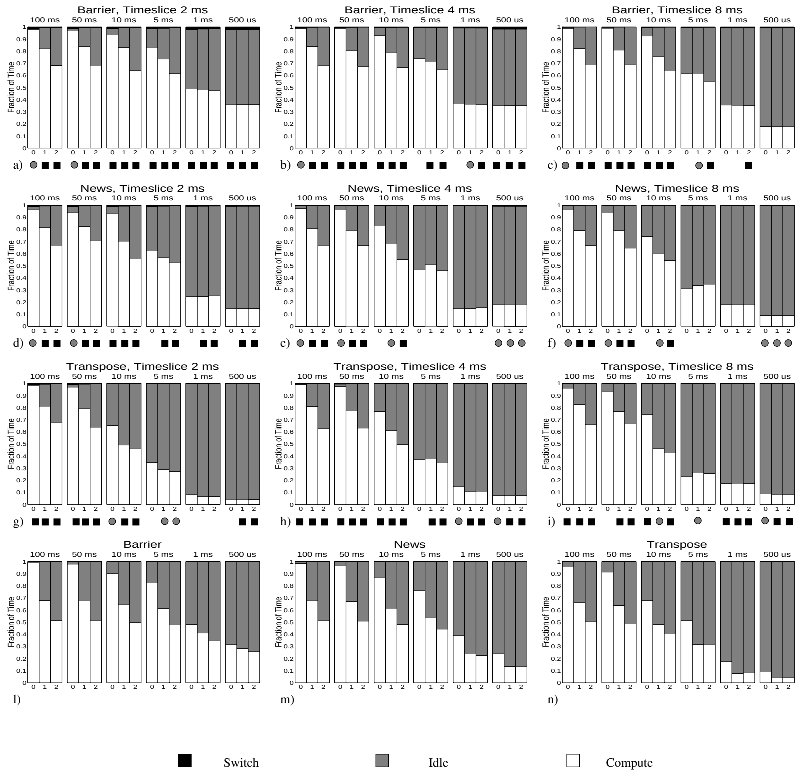

Figure 4. Resource Utilization on a Switch-Based Network.

gle job only 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.

As a final note, our preliminary experimental results do not account for the effects of the memory hierarchy on the working sets of different jobs. As a consequence, buffered coscheduling requires a larger main memory in order to avoid memory swapping. We consider this as the main limitation of our approach.

4. Conclusion

In this paper, we presented buffered coscheduling, a new methodology for multitasking jobs in parallel and distributed systems. This methodology significantly improves resource utilization when compared to existing work reported in the literature. It also 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 three families of synthetic workloads — *Barrier*, *News*, and *Transpose* — and two types of networks — a high-performance Myrinet-based network and a commodity switch-based network. Our experimental results showed that our methodology can provide better resource utilization, particularly in the presence of load imbalance, communication-intensive jobs, or a commodity network.

In the future, we intend to examine the throughput and response time of parallel jobs when using buffered coscheduling and then comparing its performance to implicit coscheduling or a space-sharing commercial solution such as LSF. We will also consider the effects of the memory hierarchy in a real application rather than in synthetic workloads as presented here.

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] 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.

[5] 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.

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

[7] 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.

[8] A. Hoisie, O. Lubeck, and H. 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.

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

[10] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. 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.

[11] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. 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.

[12] W. E. W. Patrick Sobalvarro, Scott Pakin and A. 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.

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

[14] F. Petrini and W. Feng. Scheduling with Global Information in Distributed Systems. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'00)*, April 2000.

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

[16] P. Sobalvarro and W. 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.

[17] K. Suzaki and D. Walsh. Implementing the Combination of Time Sharing and Space Sharing on AP/Linux. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1998.