SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters

L. Peng, W.F. Wong, M.D. Feng and C.K. Yuen
Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
{pengl,wongwf,fengmd,yuenck}@comp.nus.edu.sg

Abstract

Multithreaded parallel system with software Distributed Shared Memory (DSM) is an attractive direction in cluster computing. In these systems, distributing workloads and keeping the shared memory operations efficient are critical issues. Distributed Cilk (Cilk 5.1) is a multithreaded runtime system for SMP clusters with the support of divide-and-conquer programming paradigm. However, there is no support for user level shared memory. In this paper, we describe SilkRoad, an extension of distributed Cilk, which implementing the Lazy Release Consistency (LRC) memory model. In the SilkRoad runtime system, the data of system control information (such as thread management, load balancing, etc) are kept consistent by means of the backing store, just as it is in the original distributed Cilk, while the user's cluster wide shared data are kept consistent by LRC. With LRC, SilkRoad programmers are allowed to define and use shared variables between the threads running on different nodes in a cluster, and this greatly enlarged the scope of supported programming paradigms in Cilk. The result is a system that supports work-stealing and a true shared memory programming paradigm. To show the benefits of integration, we compared SilkRoad with the original distributed Cilk. We also compared SilkRoad with TreadMarks, a LRC software DSM implementation for clusters with no support of multithreading. The results show that with the hybrid memory model of dag-consistency and LRC, multithreaded SilkRoad programs written in a divide-and-conquer fashion with good data locality can achieve comparable performance with the corresponding multiple-process TreadMarks programs.

Keywords: cluster computing, multithreading, lazy release consistency.

1. Introduction

Multithreading is popular shared memory programming paradigm for symmetric multiprocessor (SMP) machines [9]. Many programming environments for cluster computing, however, support only operating system process-oriented parallelism, such as PVM [7], and sometimes are only restrictive static parallelism as is the case for MPI [8]. It is considered hard to achieve parallelism dynamically, especially with load balancing and dynamic thread creation. Nevertheless, as clusters of SMPs become prevalent, supporting multithreading in the cluster environment is desirable. With such a system, multithreaded programs written for a uniprocessor may be run on a uniprocessor, a true SMP, a cluster of uniprocessors, or a cluster of SMPs. This flexibility also promises performance scalability.

Software DSMs [15] provides the runtime support for a virtual shared memory environment over a cluster of workstations. Lazy release consistency [14] hass proved to be one of the most efficient memory consistency models in current software DSM systems. By delaying the propagation of modifications from one node to another until the next mutex acquisition, it greatly reduces communication cost.

Distributed Cilk is a multithreaded programming

systems for clusters [5] of Unix PCs. Distributed Cilk embodies the algorithmic multithreading programming language Cilk [4] and implements the language for a network of SMP's. Cilk supports "normalized"[1] *thread spawning and synchronization. and provides a limited form of distributed shared memory support for threads in the cluster. Unfortunately, user-level shared memory which is necessary in many multithreaded applications is absent in the Cilk. In the distributed Cilk run-time system, data are kept consistent by a means of a backing store. We have extended distributed Cilk by implementing LRC to support user level shared variables, allowing threads in the cluster to be interacted with the cluster wide locks. The result is a system we called* SilkRoad. *The performance of SilkRoad was evaluated by comparing it against distributed Cilk extended with straightforward user level locks, and TreadMarks [10], a popular DSM runtime system implementing LRC on cluster supporting process-oriented static multitasking using three benchmarks. Our results show that SilkRoad performs better than distributed Cilk and is comparable with TreadMarks. For programs written in divide-and-conquer approach, SilkRoad outperforms TreadMarks in some test cases.*

This paper are organized as follows: First, we will briefly describe distributed Cilk and our extension, i.e. SilkRoad, in Section 2. In Section 3, we describe the LRC protocol and its implementation in SilkRoad. Then, we describe our testbed, the applications used in the performance evaluation, and the results of the evaluation in Section 4 and Section 5. Section 6 discusses some related work. We conclude the paper with a short discussion of our future plans in Section 7.

2. Distributed Cilk

Distributed Cilk implements the basic features of the Cilk multithreaded language, which in turn is based on C language. The two basic parallel control constructs introduced in Cilk are `spawn` *for thread spawning, and* `sync` *for synchronization. The parallel control flow of a Cilk program can be viewed as a directed acyclic graph (*dag*), as illustrated in Figure 1. The vertices of the dag represent parallel control constructs, and the edges represent Cilk threads, which are maximal sequences of instructions without containing any parallel control constructs.*

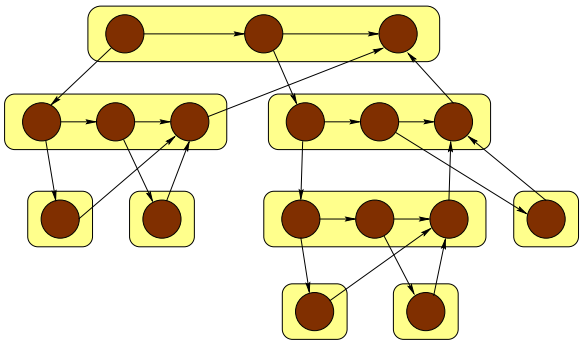

Figure 1. The parallel control flow of the Cilk program viewed as a dag.

Cilk is effective at exploiting dynamic, highly asynchronous parallelism, which may be difficult to be achieved in the data-parallel or message-passing style. For example, the divide-and-conquer paradigm can be easily expressed in Cilk: threads are dynamically spawned at the dividing stage, and synchronized at the conquering stage. There is no theoretical limit on either the number of threads that may be spawned (in practice, this is limited by the available heap space) or on the number of the nesting level of threads to be spawned (limited by the available stack space).

The Cilk run-time system uses the work stealing strategy for load balancing [3]. When a processor becomes idle, it initiate attempts to steal work (i.e., threads) from a randomly chosen busy processor. This kind of scheduler is also called greedy scheduler. *The execution time of a multithreaded program running on P processors with the greedy scheduler is $T_p \leq T_1/P + T_\infty$, where T_1 is the executing time on one processor and T_∞ is the executing time on infinite processors [6]. The work stealing strategy is also used by distributed Cilk for scheduling in a cluster of SMPs.*

As is required in the distributed memory clustering environment, the distributed Cilk runtime system implements its own distributed shared memory, supporting a memory consistency model called dag-consistent shared memory *[2]. In this consistency model, a read*

[1] Normalization means that (1) a thread which can only be joined by its immediate parent thread; (2) a parent thread will join all its created child threads before the completion. The thread relation graph (i.e. parallel control flow) is a serial-parallel graph [17].

can see the result of write only if there is a serial execution order within the dag in which the write precedes the read (i.e., the write should be executed before the read in any possible of the scheduling). This relaxed consistency model is adequate for programs written in the divide-and-conquer paradigm.

The careful reader would have noted that the inability for incomparable nodes in the dag (possibly sibling nodes) to share data is more restrictive than true shared memory processing.

To maintain dag consistency, the BACKER coherence algorithm is employed by Cilk. In this algorithm, a backing store *provides global storage for each shared object. The backing store actually consists of portions of each processor's main memory. Three basic operations, namely fetch, reconcile and flush, are used by the BACKER algorithm to manipulate shared-memory objects.*

However, distributed Cilk does not support any user-level lock as the concept of locking is absent in the dag-consistency model. Thus some applications that require locks to protect critical sections cannot run in distributed Cilk[2]*. We extended distributed Cilk by implementing cluster-wide distributed locks [13].*

We implemented distributed locking by using a straightforward centralized scheme. For each lock, a processor is chosen statically in a round-robin manner to be its manager. To obtain a lock, the acquirer will send a lock request message to the lock's manager. If no other thread is holding the lock, the manager sends a reply message to the acquirer granting the lock acquisition request. If the lock is already held by some other thread, the current acquirer waits in a queue associated with the lock. A lock holder will send a message to the manager when it releases the lock. If there are more than one acquirers waiting for the lock, the first one in the waiting queue is given the lock. The others remain in the queue. In conforming with the messaging convention in distributed Cilk, we used active messages in the sending of messages [18].

3. Lazy Release Consistency and SilkRoad

Memory consistency model defines the semantics of the shared data, namely, when the modified data on one node will be seen by the other nodes. Sequencial consistency [12] is a straightforward consistency model, but it is too strict and hard to implement efficiently. Relaxed consistency models, such as release consistency [11], were later proposed to overcome this problem. TreadMarks implements release consistency in a "lazy" way, thus its name lazy release consistency *(LRC). In LRC, each processor delays the propagation of its modifications (i.e. diffs) until the next lock acquisition.*

As mentioned in Section 2, in distributed Cilk the runtime system uses a backing store to maintain memory consistency. This causes a performance problem when user level locks are introduced: each time when there is a lock release, diffs will be created and sent to the backing store. At each lock acquire, the processor will obtain fresh diffs from the backing store by flushing its own locally cached pages. Thus the backing store is a home for all cached pages, but it is just too eager in propagating modifications.

To address this problem, we introduce LRC into distributed Cilk. We named the resulting system SilkRoad. *In the SilkRoad run-time system, all data are divided into two parts: system information (which includes thread spawning, the scheduling info, work stealing messages, etc) and the user's shared data (which is defined by the programmer). For system information, the original BACKER algorithm was used to maintain consistency between the nodes, while LRC was used to handle the consistency of the user's shared data. We opted for eager diff creation and the write invalidation protocol to propagate the modifications. User programs have to acquire cluster-wide locks to access the shared variables and then release it afterwards. When releasing a lock, the diffs for the modifications done to shared pages during this lock are created and stored. Thus there is a correspondence between diffs and locks. During the next remote lock acquisition, write notices will be sent to the acquirer. When the acquirer requests for the diffs of a page, only the diffs associated with this lock will be sent to the acquirer. So in this the number of diffs are greatly reduced.*

We measured the average time for acquiring of a lock and found it to be approximately 0.38 msec for our testbed described below.

[2]User-level lock is supported in the SMP version of Cilk.

4. Performance of the SilkRoad

In this section, we will first describe the cluster on which we run our experiments and the three application programs chosen for the experiment. This is followed by a discussion of the results.

The testbed of our experiment is a 8-node SMP PC cluster. Each node has two Pentium-III 500 MHz CPUs, 256 MB memory (512 MB for the node acting as the NFS/NIS server), and a 100Mbps Fast Ethernet network card. Nodes are interconnected in a star topology through a 100baseT switch. The operating system of each node is RedHat Linux 6.1 with the kernel version 2.2.12-20.

In our tests, the following three applications were used:

matmul *Matrix multiplication is a basic application which is widely used in benchmarking.* Matmul *multiplies two $n \times n$ matrices and puts the results into another matrix. It fits into the divide-and-conquer paradigm well: recursively split the problem into 8 $n/2 \times n/2$ matrix multiplication subproblems and combine the results with one $n \times n$ addition. This program needs the DSM support because three matrices are shared among spawned threads. No lock is needed however as the basic parallel control constructs suffice.*

queen *The objective of the* queen *program is to place n queens on an $n \times n$ chess board such that they do not attack each other. The program finds the number of all such configuration for a given chess board size. The SilkRoad program explores the different columns of a row in parallel, using a divide-and-conquer strategy. The chess board is placed in the distributed shared memory such that child threads can get the chess board configuration from their parent thread. Again user lock is not necessary in the program.*

tsp *The program* tsp *solves the traveling salesman problem using a branch and bound algorithm. In this program, a number of workers (i.e., threads) are spawned to explore different paths. The actual number of workers depends on the number of available processors. The emerged unexplored paths are stored in a global priority queue in the distributed shared memory. All workers retrieve the paths from the priority queue. The bound is also kept in the distributed shared memory, and each thread accesses (i.e., reads or writes) the bound thorough a lock, in order to ensure the consistency. Three example cases were tested: two of them with 18 cities, and one with 19 cities.*

The speedups of SilkRoad programs are listed in Table 1. These speedups are computed by dividing the sequential program's executing time by the corresponding parallel program's executing time. We used the `egcs` *compiler (version 1.1.2) with the* `-O` *option to compile all of the application programs. To run the parallel version of the program, it is copied to all the nodes involved in the computation. Where possible, we avoided using the physical shared memory of a node so as to observe the performance of the distributed shared memory. For instance, when running an instance of an application with only two computation threads, we distributed the threads to distinct nodes to minimize physical sharing.*

As expected, speedups varies depending on the applications[3]*. For the* matmul *and* queen *problems, SilkRoad achieves good speedups, especially when the problem size is large. Good speedups were also achieved for* tsp.

matmul

In this application, the divide-and-conquer strategy used in the SilkRoad program achieved good performance. For the smaller matrices (512×512), the speedup was 1.51 on two processors, while not very good on more processors because the communication overhead cannot be offset by the parallelism. The amount of concurrent work cannot make all processors busy all the time. For the larger matrices (1024×1024 and 2048×2048), we achieved good speedups and even super-linear speedups. For example, speedup 3.13 for 1024×1024 matrices on 2 processors, and speedup 3.86 for 2048×2048 matrices on 4 processors. The super-linear speedup comes from the data locality. In SilkRoad, if all elements of a divided matmul *block can fit in the local cache, there are much*

[3] *Matmul* for $n = 2048$ on 8 processors failed to run due to insufficient heap space.

Applications		2 processors	4 processors	8 processors
matmul	512×512	1.51	1.76	1.72
	1024×1024	3.13	2.67	3.18
	2048×2048	3.53	3.86	-
queen	12	2.16	4.01	6.03
	13	2.30	4.45	8.65
	14	1.70	3.41	6.65
tsp	18	1.31	2.05	1.80
	18b	1.52	3.46	3.00
	19b	1.27	1.69	1.54

Table 1. Speedups of the applications.

fewer cache misses in comparison with the sequential program that stores the matrices in the cache in row major order. When the matrices cannot fit into the local cache, thrashing occurs. On the other hand, in the SilkRoad matmul *program, the matrices are divided into small blocks till the size of* 16×16 *which fits into the local cache easily. If a thread is stolen and run on a remote processor, the amount of transferred matrices data via DSM may still be considerable, so too the amount of messages (please see Table 5 in Section 5). However, increased parallelism coupled with the data locality still tipped the balance in the favour of SilkRoad.*

queen

In this application, the SilkRoad program also uses the divide-and-conquer strategy. When the problem size increases (e.g., 14-queen problem), near linear speedups were achieved. The chess board is stored in the distributed shared memory, but the amount of data (i.e., the current chess board configuration) to be transferred is less than that of matmul. *Thus, the parallel execution did not suffer too much from the DSM overhead, and reasonable speedup is achieved through the parallelism in the problem. As is usual in parallelizing search problems, super-linear speedups were observed in some cases.*

tsp

In this application, the distance of all cities, the current shortest route found so far, the bound of the current shortest route, and a priority queue storing all unexplored routes are held in global shared memory that is frequently accessed by multiple worker threads.

5. Comparison with TreadMarks

In this section, we compare the performance of the SilkRoad with TreadMarks on the same applications we used in Section 4. TreadMarks is a typical DSM implementation for clusters without the support of multithreading. The purpose of our comparison is to the overheads in the SilkRoad run-time system as compared with a well established LRC system.

We used TreadMarks version 1.0.3 and ported it to the Linux kernel version 2.2.15-20. For the matmul *problem, we developed a corresponding TreadMarks program that statically partitions the matrices. The TreadMarks and SilkRoad* queen *programs were essentially the same. For the* tsp *problem, we used the program included in the TreadMarks distribution, and on which our SilkRoad version was based. We compile the TreadMarks runtime system code and the test applications by using the same C compiler and the optimization flags that were used for SilkRoad.*

Table 2 shows the speedups of the applications running on 2, 4, and 8 processors for both SilkRoad and TreadMarks. We observed that for matmul *(*1024×1024*), the speedup of the SilkRoad program does not increase a lot when the number of processors increases. We attribute this to the lack of parallelism. From Table 1, we see that as problem size increases, the SilkRoad programs achieve better speedups. For the* queen *program, SilkRoad's performance is com-*

parable with TreadMarks. In solving tsp *problem, SilkRoad is a little slower than TreadMarks. We believe the shared memory run-time in SilkRoad still needs to be optimized further. Section 4.*

Besides the running time and speedup, we are also interested in whether the load is balanced within clusters, and the cost of the synchronization and communication. Load balancing is a key issues in cluster computing. All things being equal, a properly load balanced system will achieve higher performance and better efficiency. We compared the load situation of TreadMarks and SilkRoad. Table 3 and Table 4 give part of the output for running matmul *(*1024×1024*) on 4 processors in SilkRoad and TreadMarks respectively. Though the data in these two tables are not directly comparable, it does show evidence that the balance of load differs in the two systems.*

Summary of time spent by each processor			
Proc. No.	Working	Total	Ratio
0	13.9000	19.4000	71.6%
1	13.3500	17.8900	74.6%
2	14.3800	20.7700	69.2%
3	14.9100	20.2200	73.7%
AVE.	14.1350	19.5700	72.2%

Table 3. Load balance in one execution of *matmul* **(**1024×1024**) on 4 processors in SilkRoad.**

processor	messages	diffs	twins	barrier waiting time (seconds)
0	7274	0	4	1.3
1	3593	256	256	1.61
2	3530	256	256	0.49
3	5838	256	256	0.49

Table 4. Load balance in one execution of *matmul* **(**1024×1024**) on 4 processors in TreadMarks.**

We can see that the load is more balanced in SilkRoad than in TreadMarks. The column under the heading "Working" reflects the time spent on executing threads in each processor respectively. The "Total" column indicates the time including the working, spawning child threads, synchronization etc. The load of the each processor in SilkRoad is roughly equally distributed as shown in Table 3. This is mainly due to the dynamic greedy scheduler. TreadMarks' static load balancing strategy is unable to maintain a balanced workload among the processors at runtime. From Table 4, we see that processor 0 receives many more messages than the other processors, while creating fewer diffs and twins during computation. The barrier waiting time also varies among the processors. One can deduce that the workload between these four processors are not well balanced.

Besides dynamic load balancing, our current SilkRoad implementation installs signal handlers for incoming messages such that incoming messages trigger signals to interrupt the working process and force it to handle I/O promptly. This works better than creating a communicating daemon process on each processor. Table 5 shows the amount of transferred data and messages in communication. We can see that during the computation, SilkRoad sends overwhelmingly more messages and transfers much more data than TreadMarks. For example, for matmul *(*1024×1024*) running on 4 processors, even though there are fewer cache misses because of the locality and small block size, SilkRoad still sends about 7.6 times more messages and transfers 4.2 times more data than the TreadMarks. We believe that the lazy release memory consistency (LRC) implemented in TreadMarks effectively reduces the amount of communication. LRC delays the propagation of the consistency information until the next time of an lock acquire operation starts, but SilkRoad also uses the backing store to maintain the consistency of the system information and this results in the large number of messages and data. Moreover, in SilkRoad, there are frequent thread migrations between processors due to the work stealing algorithm. Thread migration may trigger more DSM operations when threads access shared data structure not present in the local cache.*

Table 6 shows the time spent by the applications in synchronization between the processors in the cluster. SilkRoad takes more time in acquiring distributed locks. As shown in Table 6, the accumulative lock acquiring time in TSP in SilkRoad is about 3.7 times more than that in TreadMarks. This is mainly because that in the TSP program, some threads repeatedly ac-

Applications	No. of processors	Speedups (dis. Cilk)	Speedups (TreadMarks)
matmul (1024×1024)	2	3.13	1.28
	4	2.67	2.41
	8	3.18	4.30
queen (14)	2	1.70	2.19
	4	3.41	3.50
	8	6.65	6.65
tsp (18b)	2	1.52	1.89
	4	2.39	2.74
	8	3.00	3.21

Table 2. Speedups of the applications for both distributed Cilk and TreadMarks.

Applications	Number of messages		Transferred data (in KB)	
	dist. Cilk	TreadMarks	dist. Cilk	TreadMarks
matmul (1024×1024)	189438	24928	156016	37049
queen (12)	3577	45	1559	13
tsp (18b)	7350	5123	2635	806

Table 5. Messages and transferred data in the execution of applications (running on 4 processors).

quire and release the same lock during the computation. With the eager diff creation in SilkRoad, modifications will be saved each time when the lock is released, while in TreadMarks, lazy diff creation avoid this overhead in this case, hence the less lock acquiring time. Eager diff creation in SilkRoad associates the diffs with a particular lock which avoids send unnecessary diffs of a page. However the cost is paid in terms of the frequent diff creations in lock release.

Overall, we see that SilkRoad achieves good performance for those problems that can be solved by the divide-and-conquer strategy with little or no data dependence among the child threads, as exemplified in matmul *and* queen *programs. Even in cases where there are some increase in synchronization and communication cost, the integration of multithreading and software DSM still seems viable and good performance on a cluster is still achievable.*

The original distributed Cilk is suitable for those dynamic and highly asynchronous parallelism (such as queen*). TreadMarks is suitable for the phase parallel, or master-slave applications such as* tsp*. When dealing with some recursive problems (such as quicksort), it is more natural to choose the dynamic multithreaded programming system like SilkRoad. Certain applications (such as matrix multiplication) can be as efficient done in both programming paradigm. We have shown that by utilizing a more relaxed and efficient DSM consistency model, the dynamic multithreaded run-time system can still perform well even as it supports a wider range of parallel programming paradigms. In addition, load balancing is achieved by the greedy work stealing algorithm.*

6. Related Work

Keith Randall is the original implementor of the distributed Cilk. In his PhD thesis [16], he discussed distributed Cilk in one of the chapters but no detailed performance results except for a simple fibonacci program was given. The distributed Cilk used in this paper differs from the original distributed Cilk by Randall in two aspects: the use of signal handler to handle incoming message, which was added by Mike Bernstein of Yale University, and the provision of cluster-wide lock added by us.

There are few runtime systems for clusters which supports both load balancing and distributed shared

Lock	SilkRoad	TreadMarks
Average execution time of lock operations	0.492 msec	0.335 msec
Total time in lock acquisition for *tsp* (18b)	0.33 sec	0.09 sec

Table 6. Synchronization costs (on 4 processors).

memory. Most of parallel programming environments for cluster computing only support the static parallelism, e.g., [7, 8, 10], without dynamic load balancing. Mosix [1] supports the migration of processes for load balancing in a cluster, but lacks distributed shared memory. Much work still needs to be done in order to make clusters appear as a single system to users.

7. Conclusion and Future Work

In this paper we have described SilkRoad, an enhancement of the distributed Cilk system supporting user level global locks and a software distributed shared memory while retaining its original novel feature of multithreaded, divide-and-conquer programming paradigm coupled with work stealing.

With LRC implentation, SilkRoad performs better for those applications using user level shared variables. At the moment, it is still a hybrid shared memory system in which dag-consistency and LRC co-exist. This hybrid memory model is supports a wider range of parallel programming paradigms, but we believe its performance can still be improved. We are currently working on closing the performance gap between SilkRoad and a full LRC system like TreadMarks that still exist in some applications. Our goal is to come up with an efficient cluster run-time system that supports multithreading, load balancing and shared memory programming.

Acknowledgments

We would like to thank Willy Zwaenepoel of Rice University, Charles Leiserson of MIT for their suggestions and insightful discussions on the problems we presented in the paper.

References

[1] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. Journal of Future Generation Computer Systems*, 3(13(4-5)):361–372, 1998.*

[2] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 132–141, Honolulu, Hawaii, Apr. 1996.*

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, Nov. 1994.*

[4] Cilk-5.2 Reference Manual. Available on the Internet from `http://supertech.lcs.mit.edu/cilk`*.*

[5] Distributed Cilk - Release 5.1 alpha 1. Available on the Internet from `http://supertech.lcs.mit.edu/cilk/release/distcilk5.1.html`*.*

[6] M. Frigo, K. H. Randall, and C. E. Leiserson. The implementation of the Cilk-5 multithreaded language. In Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.*

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine. *MIT Press, 1994. PVM homepage* `http://www.epm.ornl.gov/pvm/pvm_home.html`*.*

[8] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. MPI: The Complete Reference. Volume 2 – The MPI-2 Extensions. *MIT Press, 1998. MPI forum homepage* `http://www.mpi-forum.org`*.*

[9] IEEE. Information technology–Portable Operating System Interface (POSIX)-Part1: System Application: Program Interface (API) [C Language], 1996. ANSI/IEEE Std 1003.1, 1996 Edition.

[10] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared

memory on standard workstations and operating systems. In USENIX Winter 1994 Conference Proceedings, *pages 115–132, San Francisco, California, Jan. 1994.*

[11] K.Gharachorloo, D.E.Lenoski, J.Laudon, P.Gibbons, A.Gupta, and J.L.Hennessy. Memory consistency an event ordering in scalable shared-memory multiprocessors. In Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90), *pages 15–26, May 1990.*

[12] L.Lamport. How to make a multiprocessor computer that correctly executes multiproces programs? In IEEE Transactions on Computers, *pages 690–691, Sept. 1979.*

[13] L. Peng, M. Feng, and C.-K. Yuen. Evaluation of the pervormance of multithreaed cilk runtime system on smp clusters. In Proc. of the IEEE International Workshop of Cluster Computing, *Dec. 1999.*

[14] P.Keleher, A.L.Cox, and W.Zwaenepoel. Lazy release consistency for software distributed shared memory. In Proc. of the 19th Anaual International Symposium on Computer Architecture (ISCA'92), *May 1992.*

[15] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: concepts and systems. *IEEE Computer Society, 1997.*

[16] K. H. Randall. Cilk: Efficient Multithreaded Computing. *PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998. Available as MIT Technical Report MIT/LCS/TR-749.*

[17] J. Valdes. Parsing Flowcharts and Series-Parallel Graphs. *PhD thesis, Stanford University, December 1978. STAN-CS-78-682.*

[18] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA), *pages 256–266, Gold Coast, Australia, May 1992.*