The Performance Model of SilkRoad - A Multithreaded DSM System for Clusters

Liang Peng, Weng-Fai Wong, and Chung-Kwong Yuen
Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
{penglian,wongwf,yuenck}@comp.nus.edu.sg

Abstract

Distributed Shared Memory (DSM) is a highly desirable programming model for cluster based computing. Even though a number of software DSMs have been developed with their performance evaluated, few of them have a theoretical performance model. In this paper, we propose and analyze the performance model of the software DSM of SilkRoad, a multithreaded runtime system for cluster computing. SilkRoad is built on the Cilk system with an extended memory consistency model which we call RC$_{dag}$ *consistency. Extending Cilk's theoretical performance model, we show that with the RC*$_{dag}$ *consistent DSM, the expected execution time* T_P *of a partially strict multithreaded computation on* P *processors is* $O((T_1(Z,L)+N)/P+\mu H T_\infty)$*, where* $T_1(Z,L)$ *is the total work of computation,* T_∞ *is the critical path,* N *is the number of lock acquisitions,* μ *is the service time of a cache miss, and* H *is the height of memory. Finally, we present experimental evidence that verify the performance model.*

Keywords: *Software Distributed Shared Memory, memory consistency models, theoretical performance model.*

1. Introduction

There are many memory consistency models and parallel programming systems that model the behavior of memory and processor in parallel architectures. Some of them focus on specifying what happens when a processor performs some operations on memory and hence they are called *processor-centric* memory models. While others focus on the computation and not on the schedule, and hence are called *computation-centric* memory models[9]. The philosophy of the computation-centric approach is to separate the logical dependencies between instructions (the computation) from the way that instructions are mapped to the processors (the schedule) [8]. One example of a computation-centric memory model is the *Dag Consistency or Location Consistency* model[8, 4, 3] that was developed for Cilk-like multithreaded computations[6, 7, 10, 14]. Cilk uses a randomized work-stealing scheduler and achieves performance close to the lower bounds of *fully strict* multithreaded algorithms without using user-level shared memory. In a fully strict multithreaded algorithms a thread may only synchronize with its children. In Cilk, a multithreaded program defines a partial execution order on its instructions and the partial order can be viewed as a directed acyclic graph (*dag*). In the theory of fully strict multithreaded computation, the *work* of computation, denoted T_1, is the number of instructions in the *dag*. This corresponds to the amount of execution time on a single processor. The *critical-path length* of the computation, denoted T_∞, is the maximum number of instructions on any directed path in the *dag*. This corresponds to the amount of execution time required on a system with an infinite number of processors. Specifically, for any such algorithms and any P number of processors, the randomized work-stealing scheduler (which is maintained by an algorithm called the BACKER algorithm[4]) executes the algorithm in expected time $T_P = O((T_1(Z,L))/P + \mu H T_\infty)$ [8], where $T_1(Z,L)$ is the total work of the computation, μ is the latency of a page fault, Z is the cache size, and L is the cache line size, $H = Z/L$ is the height of the cache.

The above result is the analysis of the execution time of "fully strict" multithreaded algorithms that use Location Consistent shared memory. A multithreaded computation is fully strict if every dependency edge goes from a procedure to either itself or its parent procedure [5, 8]. However, for the multithreaded computations which are not fully strict, the situation is complex because the data dependency and synchronization may also happen between incomparable children. In this paper we extend the memory model and its performance bounds to that for *partially strict* multithreaded computation by considering the synchronizations between sibling threads. This is a mode of programming that is commonly found in parallel multithreaded computations. This extension significantly enlarges the applicability of computation-centric memory model. We will also show a way to generalize our analysis results. In this paper, we will only consider synchronization with lock-like mutual exclusion, which can be modeled as *(acquire,release)* pairs, and the resultant extension of the *dag* of a computation. Our previous work formally defined a new memory consistency model, i.e. RC_{dag} consistency, on the basis of the extended *dag*. Given the above *dag* computation and the memory model, we proposed a *stealing based coherence* algorithm. In this paper, we show that for the partially strict multithreaded computation executed on P processors, using the work-stealing scheduler in conjunction with the stealing based coherence algorithm, the expected execution time is $O((T_1(Z,L)+N)/P+\mu HT_\infty)$, where N is the number of lock acquisitions in the computation.

The rest part of this paper is organized as follows. In Section 2, the RC_{dag} consistent memory model of SilkRoad is introduced. In Section 3, we propose and analyze the performance model of the DSM in SilkRoad. Section 4 give some experimental results and the discussion on the performance. Some related work are also introduced in Section 5. Finally, we give a conclusion in Section 6.

2. SilkRoad and RC_{dag} consistent memory model

SilkRoad [12, 13] is a variation of Cilk. Its RC_{dag} memory consistency model is an extension of Cilk's Location Consistency (LC). One of the main difference between SilkRoad and Cilk is that SilkRoad provides a shared memory at the user programming level and does not use the backing store (which is a "home" of the virtual memory space) of Cilk at runtime system level. In SilkRoad, a concept of *stealing based coherence* (SBC) is proposed to implement RC_{dag} consistency. The SBC also uses the basic operations in Cilk: *fetch* (get the most up-to-date data from the backing store), *reconcile* (write the most up-to-data data back to the backing store), and *flush* (remove local data which is obsolete). These operations in SilkRoad are also triggered by thread stealing and return like in Cilk, but the difference is that in SBC a fetch operation copies diffs from the node (one of its ancestor) who did the modification, not from the backing store; a reconcile operation just save the diffs locally and propagates when required, instead of copying them from local cache to backing store. That is to say, in SilkRoad, the memory consistency is kept by thread stealing and return operations.

The execution of a multi-threaded computation in Cilk can be viewed as a directed acyclic graph (dag). The use

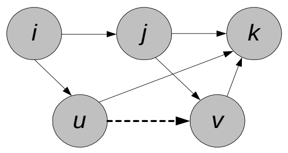

Figure 1. In the extended dag, threads can synchronize with their siblings. i, j, and k are the procedures in parent thread; u and v are procedures in child threads. The dotted arrow (u, v) is a synchronization edge.

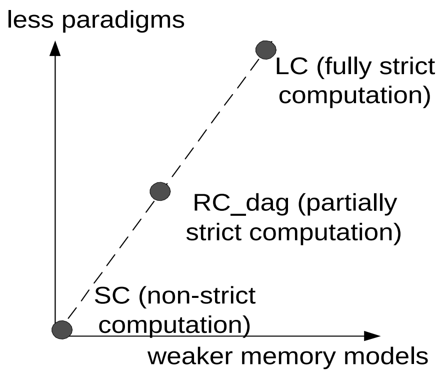
less paradigms
LC (fully strict computation)
RC_dag (partially strict computation)
SC (non-strict computation)
weaker memory models

Figure 2. The RC_dag consistency is stronger than LC but weaker than SC.

of some synchronization mechanisms to provide for control access to shared virtual memory is a popular technique in parallel programming. Unfortunately, this is not supported in Cilk. Cilk's computation dag is extended in SilkRoad by the introduction of locks, i.e. (release, acquire) pairs. A lock can be modeled as a *synchronization edge* and thus extending the dag of the computation (Figure 1). In RC_{dag} consistency, the values can be transferred not only via the threading stealing/return, but also via the global synchronization like lock acquiring/releasing.

Along with extending the dag, the corresponding memory consistency model is also changed. The RC_{dag} consistency is developed based on the LC consistency model under the general computation centric theory. RC_{dag} consistency is a more stringent memory consistency model than the LC consistency, but it is weaker than Sequential Consistency (SC). Their relationship is shown in Figure 2.

In order to support more computation by extending the dag of computation, the semantics of *lazy release consistency* (LRC) [11] are introduced into location consistency model. LRC is tightly coupled with some synchronization mechanisms such as locks and barriers. So introducing LRC means that the dag of computation will be extended to include the features of mutual exclusion and global synchronization. With these extensions, the computation is no longer fully strict since the siblings in the dag can also interact through synchronization edges. We call this *partially strict computation*.

3. Analysis of Execution Time

In software DSM systems, the execution time of applications consists of the following constituents: computation time, scheduling overhead, and synchronization overhead. Computation time is the actual time spent in computing, and this is determined by the application and the hardware of the nodes. The scheduling overhead is the overhead in distributing computation to each node. In the systems with dynamic scheduling (such as Cilk and SilkRoad), the scheduling overhead exists throughout the entire length of an application's execution. The synchronization overhead is the overhead in performing synchronization operations, for example, lock acquisition for a critical section or barrier synchronization. In the $\mathrm{RC}_{\mathrm{dag}}$ consistency, the synchronization overhead is tractable because of the semantic property of the release consistency: the maintenance of consistency of data is delayed until *releases* and *acquires* occur. The total execution time T_P of an application running on a software DSM system with P nodes can be expressed as follows:

$$T_P = T_C + T_S + T_{\mathrm{syn}}$$

where T_C is the computation time, T_S is the scheduling overhead, and T_{syn} is the overhead because of global synchronization.

In Cilk, for a multi-threaded computation which has T_1 total work (the execution time on one processor) and T_∞ critical-path length (the execution time on infinite number of processors), the expected execution time on P processor is $O(T_1(Z,L)/P + \mu H T_\infty)$, where Z is cache size, L is the line size of the cache, $H = Z/L$ is the cache height, and μ is the service time for a cache miss without congestion [8, 3]. T_C and T_S are already found in Cilk's model. Since Cilk does not support global mutual exclusion, the T_{syn} portion does not exist in Cilk. With the extended *dag*, for the partially strict computation, the T_{syn} part should be considered in SilkRoad because there may be global synchronization between threads.

In this section, we bound the execution time of partially strict multithreaded algorithms with a $\mathrm{RC}_{\mathrm{dag}}$ consistent memory model when the parallel execution is scheduled by the work-stealing scheduler and the $\mathrm{RC}_{\mathrm{dag}}$ consistency is maintained by the Stealing Based Coherence algorithm. We employ the terminologies introduced by Frigo [8]. Specifically, for a given partially strict multithreaded algorithm, let Z denotes the size of local cache with the line size of L and height of $H = Z/L$. Let μ be the service time for a cache miss assuming no network congestion. Suppose computation has T_1 computational work, $Q(Z,L)$ serial cache misses, $T_1(Z,L) = T_1 + \mu Q(Z,L)$ total work, and T_∞ critical-path length. In this section, we show that the expected execution time is $O((T_1(Z,L) + N)/P + \mu H T_\infty)$, where N is the number of lock acquisitions in during the computation. The exposition of the proofs in this section makes heavy use of the results and techniques of Frigo[8].

Frigo[8, 3] proved that for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the total number of steal requests and related page transfers is at most $O(HPT_\infty + HP\lg(1/\epsilon))$. This Lemma is based on multithreading with a work-stealing scheduler, so it still holds for when the strictness of computation is relaxed in SilkRoad, because the random work-stealing scheduler is still used. This observation will be used in proving the following theorem.

Theorem 1 *Consider any partially strict multithreaded computation executed on P processors, each with an $LRU(Z,L)$-cache of height H, using the work-stealing scheduler (like in Cilk and SilkRoad) in conjunction with the Stealing Based Coherence (SBC) algorithm. Let μ be the service time for a cache miss assuming no network congestion, and assume that accesses to the main memory are random and independent. Suppose the computation has T_1 computation work, $Q(Z,L)$ serial cache misses, $T_1(Z,L) = T_1 + \mu Q(Z,L)$ total work, and T_∞ critical-path length. Then for any $\epsilon > 0$, the execution time is $O((T_1(Z,L)+N)/P+\mu H T_\infty+\mu \lg P+\mu H \lg(1/\epsilon))$ with probability at least $1 - \epsilon$, where N is the number of lock acquisitions in the computation. Moreover, the expected execution time is $O((T_1(Z,L) + N)/P + \mu H T_\infty)$.*

Proof: We shall use the same accounting argument given by Frigo[8, 3] to bound the running time. During the execution, at each time step, each processor puts a piece of silver into one particular buckets according to its activity at that time step. However, for partially strict multithreaded computation, we must consider two more buckets: LOCK and LOCKWAIT. In addition, unlike Frigo [8], since the SBC algorithm is not using the *backing store* as a shared virtual memory for the run-time system, there is a little change with the buckets XFERWAIT.

- **WORK**. A silver is put in this bucket if the processor executes a task. So this bucket contains exactly T_1 dollars, because there are exactly T_1 tasks in the computation.

- **STEAL**. A silver is put in this bucket if the processor sends a steal request. Since there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ steal requests (see Lemma 26 in Frigo's thesis[8]), there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ pieces of silver in the STEAL bucket. This portion is determined by the random work-stealing scheduler.

- **STEALWAIT**. A silver is put in this bucket if the processor waits for a response to a steal request. According to the **recycling game**[5], if N requests are distributed randomly to P processors for service, with at most P requests outstanding simultaneously, the total time waiting for the requests to complete is $O(N + P \lg P + P \lg(1/\epsilon'))$ with probability at least $1 - \epsilon'$. Since there are $O(HPT_\infty + HP \lg(1/\epsilon'))$ steals, then the total time waiting for steal requests is $O(HPT_\infty + P \lg P + HP \lg(1/\epsilon'))$ with probability at least $1 - \epsilon'$[8]. However, in this case, since for the SBC algorithm, there are no reconciliation with a *backing store* for SBC algorithm, we do not need to account for the time spent in reconciling. With the consideration of the idle steps to avoid too frequent steal requests[8, 3], the total number of silver in this bucket is $O(\mu HPT_\infty + P \lg P + HP \lg(1/\epsilon'))$.

- **XFER**. If the processor sends a line-transfer request, it puts a piece of silver into this bucket. Even though in SBC algorithm the request is sent to the last victim instead of the *backing store*, the number of pieces of silver in this bucket is still $O(\mu Q(Z, L) + \mu HPT_\infty + \mu HP \lg(1/\epsilon'))$[8].

- **XFERWAIT**. If the processor waits for a line transfer to complete, it puts a piece of silver into this bucket. The recycling game shows that there are $O(\mu Q(Z, L) + \mu HPT_\infty + \mu P \lg P + \mu HP \lg(1/\epsilon'))$ pieces of silver in this bucket with probability at least $1 - \epsilon'$.

- **LOCK**. If the processor executes an *acquire* operation, it puts a piece of silver into this bucket. This results from the extended dag for partially strict multithreaded computation. The number of lock acquisitions depends on the application and the scheduler. We may suppose there are N lock *acquire* operations.

- **LOCKWAIT**. If the processor waits for the lock request to be granted, it puts a piece of silver into this bucket. Also according to recycling game, the total waiting time for the lock acquirement is $O(N + P \lg P + P \lg(1/\epsilon'))$ with the probability of at least $1 - \epsilon'$.

Now we add up the silver in each bucket and divide it by P to get the running time. With probability at least $1 - 2\epsilon'$, the sum of all the pieces of silver in all the buckets is $T_1 + O(\mu Q(Z, L) + \mu HPT_\infty + \mu P \lg P + \mu HP \lg(1/\epsilon') + N)$ with probability at lease $1 - 2\epsilon'$. Dividing by P, we can obtain runtime $T_P \leq O((T_1 + \mu Q(Z, L))/P + \mu HT_\infty + \mu \lg P + \mu H \lg(1/\epsilon') + N/P)$ with probability at lease $1 - 2\epsilon'$. Using the identity $T_1(Z, L) = T_1 + \mu Q(Z, L)$ and substituting $\epsilon = 2\epsilon'$ yields the high-probability bound. The expected bound follows similarly. □

4. Experimental Results

4.1 Experiments

In this section we will use empirical measurements on an implementation of SilkRoad to verify the correctness of our model. The test-bed for our experiment is an 8-node PC cluster. The processor of each node is Pentium-III 500 MHz CPU. The memory size 256 MB (or 512 MB for the node acting as the NFS/NIS server). Nodes are interconnected with 100Mbps Fast Ethernet network in a star topology through a 100baseT switch. The operating system of each node is RedHat Linux 6.1 with the version 2.2.12-20 kernel.

In our tests, six applications were used: matrix multiplication, the N-queen problem, Barnes-Hut, LU decomposition, the Traveling Salesman Problem, and an embarassing parallel benchmark.

Matrix multiplication (*matmul*) is a basic application which is widely used in benchmarking. The *matmul* program multiplies two $n \times n$ matrices A and B and puts the results into another matrix C. It fits well into the divide-and-conquer paradigm: recursively splitting the problem into eight $n/2 \times n/2$ matrix multiplication subproblems and combining the results with one $n \times n$ addition. This program needs the DSM support at runtime level because the three matrices are shared among the spawned threads.

The objective of the N queen (*nqueen*) program is to place n queens on an $n \times n$ chess board such that they do not attack each other. The program finds all such configurations 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 DSM such that child threads can get the chess board configuration from their parent thread.

The *barnes-hut* program simulates the interaction of n point masses under the interaction of gravity. This simulation is used in astrophysics to explore the dynamics of galaxies and galaxy formation. In *barnes-hut*, many distant masses are approximated by a single large mass in order to reduce the number of force calculations required. In the SilkRoad, a single phase of the computation, the force calculations, is parallelized.

The *tsp* program 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. Unexplored paths are stored in a global priority queue in the DSM. All workers will retrieve the paths from the priority queue. The bound is also kept in the DSM, and each thread accesses (i.e., reads or writes) the bound through a lock, in order to ensure the consistency.

The LU program performs the divide and conquer form of a blocked LU decomposition of a dense matrix $(A = LU)$. LU factorization is the most time consuming step of a common method of solving a system of linear equations. The dense $n \times n$ matrix is divided into an $N \times N$ array of $B \times B$ blocks to exploit temporal locality of sub-matrix elements ($n = NB$). In our experimental program ***lu***, the block size is set to be 16.

The Embarrassingly Parallel (*ep*) accumulates two-dimensional statistics from a large number of Gaussian pseudo-random numbers which are generated according to a particular scheme that is well-suited for parallel computation.

Table 1 lists the experimental results.

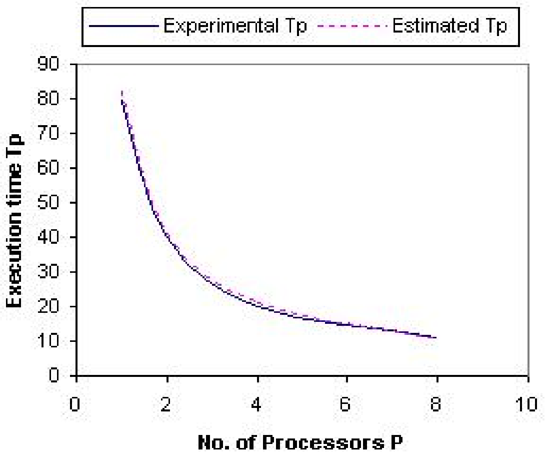

Figure 3. The performance of NQueen problem

4.2 Modeling Performance

In Section 3, we proposed the performance model of SilkRoad. Now we shall try to verify the model on the basis of the experimental data. Specifically, we use the *nqueen* and *tsp* to show that with the *work* T_1, *critical path* T_∞ and global synchronization cost $T_S(N)$ (N is the number of lockings), the runtime of an application on P processors can be approximately modeled as $T_P \approx c_1(T_1/P) + c_\infty T_\infty + PT_S(N)$ (c_1 and c_∞ are constants).

For the *nqueen* problem, since there is no global synchronization, the performance model is actually Cilk's model: $T_P = c_1(T_1/P) + c_\infty T_\infty$. For example, in 13 queen problem, T_1 and T_∞ (measured by run-time system) are 79.64s and 0.03s respectively, $T_S(N)$ is 0. According to the result data, we can calculate the constants c_1 and c_∞, which are 1.03 and 10 respectively. So the performance of 13 queen problem on P nodes can be modeled as $T_P \approx 1.03(T_1/P) + 10T_\infty$. Figure 3 shows the comparison between the estimated performance and tested performance. To measure how well the experimental results fit the predicted performance, we use the *coefficient of determination* R^2, which is defined as follows:

$$R^2 = 1 - \frac{\sum(y - y')^2}{\sum(y - \overline{y})^2}.$$

The closer R^2 is to 1, the better the prediction. Based on the experimental data in Table 1 and the predicted results (82.32s, 41.31s, 20.81s, and 10.55s on 1, 2, 4,and 8 nodes respectively), we got $R^2 = 0.996$, which is very close to 1. This shows that our proposed RC$_{\text{dag}}$ consistency does not destroy LC's initial performance model.

For the TSP 19b problem, T_1 is 11.94s, T_∞ is 2.35s, c_1 is 0.962, c_2 is 0.085, and $T_S(N)$ is 0.421. So the performance of *tsp* 19b on P nodes can be approximately modeled as $T_P \approx 0.962(T_1/P) + 0.085T_\infty + 0.421P$. However, since $T_S(N)$ (which means the overhead of locking) of *tsp* is large, it is hard to achieve good speedups. This shows that the implementation of the SilkRoad runtime system needs improvement in order to reduce $T_S(N)$. Figure 4 shows the comparison between the estimated performance and tested performance. Similarly, we calculated the coefficient of determination based on the experimental results (see Table 1) and predicted results (12.08s, 6.78s, 4.75s, and 4.99s on 1, 2, 4, 8 nodes respectively) and we got $R^2 = 0.98$.

The performance modeling provides information to analyze and evaluate the system. To some extent, it is helpful to find out where the possible bottlenecks are (for example, in computation, scheduling, or synchronization),and hence make improvements according (such as better network, faster processors, etc.).

5. Related Work

Some performance models have been proposed for DSMs. Donald Yeung et al. [15, 16] started from a clusters of SMPs with a relatively simple protocol. Blelloch and Gibbons et al. studied the performance of planar DAG scheduling. Their model supports synchronization based on write-once synchronization variables [2]. Bilas [1] analyzed the performance of shared virtual memory on networks from communication layer, protocol layer, and application layer. Our performance model is based on Cilk's initial model and the effect of DSM operations.

Applications		sequential execution	2 processors	4 processors	8 processors
matmul	512×512	9.81s	5.79s	5.03s	4.73s
	1024×1024	84.66s	38.41s	28.08s	24.75s
nqueen	12	14.64s	6.99s	3.61s	2.15s
	13	76.61s	39.94s	19.75s	10.82s
	14	528.34s	310.31s	155.03s	81.98
lu	512×512	18.16s	5.23s	4.82s	4.77s
	1024×1024	83.56s	28.28s	21.74s	16.27s
barnes-hut	16384	144.2s	112.4s	96.68s	81.53s
tsp	19b	11.58s	6.85s	5.49s	4.75s
ep	2^{24}	23.02s	11.66s	6.01s	3.15s

Table 1. Timing of the SilkRoad applications.

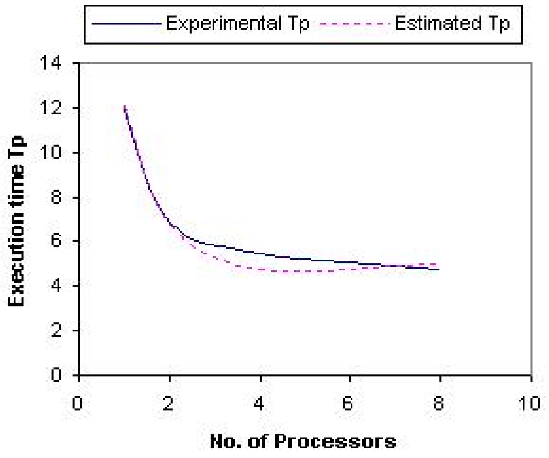

Figure 4. The performance of TSP

6. Conclusion

In this paper, we discussed the performance model of RC$_{\text{dag}}$ consistency in SilkRoad. The analysis of the performance model is based on Cilk's theoretical performance model. We showed that by extending Cilk's LC consistency to RC$_{\text{dag}}$, the expected execution time T_P of a partially strict multithreaded computation on P processors is $O((T_1(Z,L) + N)/P + \mu H T_\infty)$, where $T_1(Z,L)$ is the total work of computation, T_∞ is the critical path, N is the number of lock acquisitions. Our experiments shows the results predicted by performance model are close to the those of real execution. The model also helped us in analyzing our implementation so as to locate performance bottlenecks.

7. Acknowledgments

We would like to thank Charles Leiserson and Matteo Frigo of MIT for their suggestions and discussions on the problems we presented in the paper.

References

[1] A. Bilas. *Improving the Performance of Shared Virtual Memory on System Area Networks*. PhD thesis, Department of Computer Science, Princeton University, Nov. 1998.

[2] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 12–23, June 1997.

[3] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared Memory Algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Padua, Italy, June 1996.

[4] 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.

[5] 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.

[6] Cilk-5.2 Reference Manual. Available on the website `http://supertech.lcs.mit.edu/cilk`.

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

[8] M. Frigo. *Portable High-Performance Programs*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[9] M. Frigo and V. Luchangco. Computation-centric memory models. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1998.

[10] 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.

[11] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Electrical Engineering and Computer Science, Rice University, Jan. 1995.

[12] L. Peng, W. F. Wong, M. D. Feng, and C. K. Yuen. SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters. In *Proc. of the 2nd IEEE International Conference on Cluster Computing (CLUSTER2000)*, Nov. 2000.

[13] L. Peng, W. F. Wong, and C. K. Yuen. SilkRoad II: A Multi-Paradigm Runtime System for Cluster Computing. In *Proc. of the 4nd IEEE International Conference on Cluster Computing (CLUSTER2002)*, Sept. 2002.

[14] 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.

[15] D. Yeung. The scalability of multigrain systems. In *13th Annual International Conference on Supercomputing*, June 1999.

[16] D. Yeung, J.Kubiatowicz, and A.Agarwal. Mgs: A multigrain shared memory system. In *23th Annual Symposium on Computer Architecture*, pages 44–55, May 1996.