Smile consistency - A Memory Consistency Model with User Definable High Level Synchronization Primitives

Churngwei Chu Peter Piatko
Department of Computer Science
New York University
715 Broadway 7th Floor
New York, NY 10003
email: chu@cs.nyu.edu, piatko@cs.nyu.edu

Abstract

We propose a new natural memory consistency model, Smile consistency. Not only does Smile provide an intuitive memory consistency model but also a paradigm in which users can define their own synchronization primitives, called synchronization classes. Programmers can use the synchronization class to ease the programming work related to basic synchronization operations. Therefore, in addition to shared memory, threads can also communicate with each other via synchronization objects, instances of synchronization classes. Programs with high-level synchronization objects may also outperform those with only basic synchronization primitives.

KEYWORDS: Software Distributed Shared Memory

1 Introduction

Software distributed shared memory uses workstations and local networks to provide an economical virtual parallel machine without using any special hardware. The two main components of any distributed shared memory system are the memory consistency protocol that defines the behavior of memory when threads access it, and the synchronization operations used to synchronize threads and data access. Systems such as [LH89], [BZS93],
[CBZ95], and [ACD$^+$96] have worked at creating efficient consistency protocols to enhance data access to shared memory.

However, these systems just provide very basic synchronization primitives, for example locks and barriers. The complexity of using such basic synchronization operations to solve some synchronization problems is known to be very complicated for programmers and prone to errors[Tan92].

We introduce a new natural shared memory model that is a combination of user-definable high level synchronization primitives and a variant of release consistency. We call this model Smile consistency. User-definable synchronization primitives ease the coding of complicated parallel programs. In some cases the programmer can use a high-level synchronization primitive to increase performance by reducing unnecessary message-passing between processors.

In section 2, we will give a brief overview of the programming model of Smile. Section 3 describes the details of Smile consistency, defining the interaction between synchronization objects and shared memory. In section 4, we briefly discuss the implementation of the synchronization objects, and in section 5 we describe an application and performance results. Related work is discussed in section 6.

2 Programming Model

There are two types of shared objects that threads use to communicate with each other in Smile — shared memory and synchronization objects. Shared memory consists of a contiguous memory array in virtual memory. Two operations are allowed on shared memory, reads and writes. Whatever a thread writes on shared memory may be visible to other threads. In addition to shared memory, threads can also communicate with each other via synchronization objects. Each thread accesses synchronization objects only by calling operations defined in synchronization classes. Synchronization objects can not access other shared objects.

Each synchronization operation is annotated with one of following attributes, *put*, *get*, *get_put* and *put_get* or nothing. These attributes are used to define the visibility of the values in shared memory. Informally, the annotation *put* may be thought of as meaning that the thread "puts" its shared memory writes onto the synchronization object, and *get* meaning that the thread "gets" the previous writes from the synchronization object. For example, suppose a thread t writes to shared variable X and then performs a *put* operation of synchronization object S. If thread t' subsequently performs an *get* operation of S, t' can read what t has written on X. Similarly, *get_put* and *put_get* may be thought of as a combination of the two. The meaning of the annotations is discussed in detail in section 3.2.

2.1 Synchronization Objects and Synchronization Classes

Synchronization objects can be accessed only by calling operations provided by them. The computation of a synchronization object is like that of a server servicing a remote procedure call and is sequential. The synchronization object may decide not to reply immediately to the caller and receive other requests. The caller is stalled until the synchronization object sends a response back.

Defining synchronization classes is very similar to defining regular classes in the C++ programming language, (see Fig. 1). Instead of *class* in C++, synchronization classes start with *SyncClass*. Synchronization classes do not currently have the inheritance properties of regular object oriented languages. The operations declared in the public section of the synchronization class are the only

```
SyncClass ParaBuf_class{
public:
  ParaBuf_class(int size_of_buffer);
  put int PutItemPtr(bulk_itemptr_type itemptrs);
  get itemptr GetItemPtr();
private:
  Queue_class<int> *empty;  /* keep thread requests when the buffer is empty (or full) */
  Queue_class<itemptr_type> *buffer;  /* keep products´ pointers*/
};

ParaBuf_class *ParaBuf;

main(int argc, char **argv){
  .......
  ParaBuf = new ParaBuf_class;
  .......
}
```

Figure 1: Synchronization Class

operations which can be called by threads. Each operation can have at most one parameter. In addition, each of the public operations may be tagged with a synchronization attribute of either *put*, *get*, *put_get*, or *get_put*. These attributes are used to define the visibility of threads' writes to shared memory.

Our system provides two basic synchronization classes, semaphores and barriers. Semaphores have two operations, $P(k)$ and $V(k)$, where k is the number to increase or decrease the counter of the semaphore. $P(k)$ and $V(k)$ are annotated with *get* and *put* attributes respectively. The P and V operations of binary semaphores usually correspond to operations on locks, which are used to protect a critical section. When a thread locks a lock using the P operation, the *get* annotation specifies that the previous lock holder's writes become visible to the thread. Similarly, when the thread leaves the critical section using the V operation, the *put* annotation specifies that subsequent threads will see its writes.

A barrier has one operation, *WaitForBarrier*(*proc*), where *proc* is the number of attending threads. The attribute of *WaitForBarrier*(*proc*) is *put_get*. Intuitively, the *put_get* attribute can be thought of specifying a *put* and then a *get*. In this case, the calling thread will be "putting" its writes (i.e. making them visible) when it calls *WaitForBarrier* and will be "getting" other writes (i.e. collecting currently visible writes) when the call returns.

2.2 Shared Memory

Shared memory consists of a set of shared variables, X, Y, Z ...etc.. Without loss of generality we deem each variable an integer. A visible set of X to thread t, V_X, is a set of values thread t may get when t reads X. Initially $V_X = \{0\}$ to all threads. In the following discussion, we are going to discuss computation without race conditions at first. More details can be found in section 3.

When a thread performs a *put* operation[1] on synchronization object S, it also gives the update information on shared memory to S. When a thread performs *get* operation on a synchronization object, the thread also obtains all updates that have been put on S so far. If a thread t_1 writes a to X, the visible set of X becomes $\{a\}$. If after t_1 performs a *put* operation on synchronization object S, no other threads write to X, then if subsequently thread t_2 performs an *get* operation on S, V_X to thread t_2 is $\{a\}$.

Fig. 1 shows a fragment of a synchronization class which defines a buffer for a producer-consumer style program. In this example, the buffer stores pointers to products in shared memory. There are two operations on the synchronization class, *PutItemPtr*() and *GetItemPtr*(). *PutItemPtr*() is annotated with *put* because the update made by producers should be visible by consumers. The return value of
PutItemPtr() indicates whether the buffer is full. *GetItemPtr*() is annotated with *get* because the requesting thread needs to see what producers write. The returned value of *GetItemPtr*() is the pointer of an available product or a null pointer if empty.

3 Smile consistency

The execution of a thread or a synchronization object itself is sequential. The updates of one thread which performs a *put* operation on a synchronization object S, are visible to another thread, t, which subsequently performs *get* operation on S.

In this section, we are going to define the relation between the execution of a parallel program and the behavior of the shared memory.

3.1 Phases and Events

The execution of a thread or a synchronization object is sequential and consists of phases. Performing a synchronization operation is an event of a thread. Receiving a request from a thread and replying to a thread are events of a synchronization object. The computation between two events, including the ending event, is a phase. The execution of a thread or a synchronization object is a sequence of phases in program order. The phase of a synchronization object starting with a requesting event

[1] In the context of this discussion *put* operations are operations with attribute *put*, *get_put* or *put_get*. *Get* operations are operations with attribute *get*, *put_get*, or *get_put*.

from a thread is called a *receiving phase*, even though the requesting event is not in the phase. The phase of a synchronization object ending with a replying event to a thread is called a *replying phase*.

3.2 The Execution of a Parallel Program

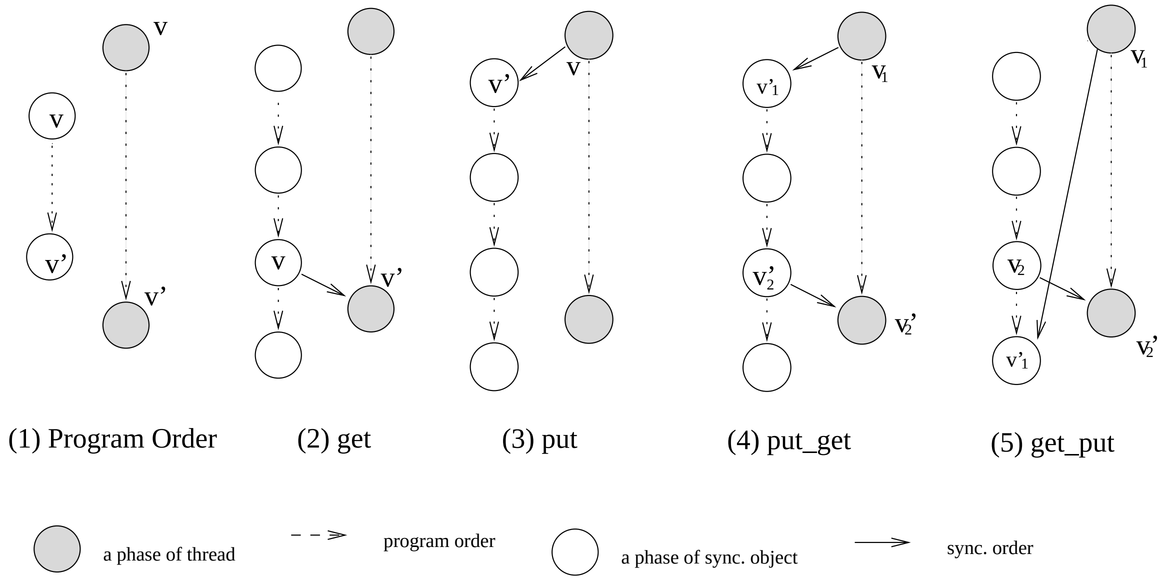

Figure 2: A directed acyclic graph of a computation.

The execution of a parallel program can be represented by a directed acyclic graph, $G = \{V, E\}$, where V is the set of phases and E is the set of edges connected by two phases. The visibility of a written value on shared memory is defined by G. There is an edge $e_{vv'}$ from vertex v to v' if and only if one of following conditions exists.

1. v' and v are phases from the execution of the same thread or synchronization object and v' immediately follows v in program order. See Fig. 2(1).
2. Thread i invokes a synchronization operation with the *get* attribute. v is the replying phase of the synchronization object, and v' is the phase after thread i invokes the synchronization operation. See Fig. 2(2).
3. Thread i invokes a synchronization operation with the *put* attribute. v is the phase ending with the synchronization operation, and v' is the receiving phase of the synchronization object. See Fig 2(3).
4. Thread i invokes a synchronization operation with the *put_get* attribute. In this case there are two edges.

   (a) v is the phase of thread i ending with the synchronization operation, and v' is the receiving phase of the synchronization object. See Fig 2(4), edge $e_{v_1 v_1'}$.
   (b) v is the replying phase of the synchronization object and v' is the phase after thread i invokes the synchronization operation. See Fig 2(4), edge $e_{v_2 v_2'}$.

5. Thread i invokes a synchronization operation with the *get_put* attribute. In this case there are two edges:

   (a) v is the phase of thread i ending with the synchronization operation, and v' is the phase immediately after the replying phase of the synchronization object. See Fig 2(5), edge $e_{v_1 v_1'}$.
   (b) v is the replying phase of the synchronization operation, and v' is the phase after thread i invokes the synchronization operation. See Fig 2(5), edge $e_{v_2 v_2'}$.

A phase p is reachable to p' if there is a path from p to p'. Two phases are concurrent if there is no path between them.

3.3 Visibility of a Written Value on a shared variable

A written value of a write operation, w, on shared variable x in phase p is visible to a read operation on x in phase p' if and only if one of following situations holds

1. if $p = p'$, the write operation is the last write operation on x before the read operation.
2. if $p \neq p'$ and p is reachable to p', w is the last write on x in phase p and there is no other phase on the path from p to p' which has write operation on x.
3. p and p' are concurrent.

The set of written values visible to the read operation *op* is called the *visible set* of *op*. Any of the values in the visible set returned for *op* is legal in Smile.

From the programmer's point of view, a *get* action happens when the synchronization object replies to the thread. All updates on shared memory visible to the synchronization object before it replies are also visible to the thread after the thread receives the response from the synchronization object.

The time *put* acts depends on how conservatively the updates on shared memory are expected to be propagated. If attribute *put_get* or *put* is used, the updates of the thread which performs the synchronization operation are visible to synchronization object when the object receives the request. If the operation is annotated with *get_put*, the updates are made visible to the synchronization object after the object replies to the thread.

A synchronization operation can be without any attribute. In such case, the updates of the thread are not visible to the synchronization object when it performs the synchronization operation.

4 Implementation of Synchronization classes and objects

In the current implementation, on each processor there are two processes in charge of the computation. One process, the thread, is used for the computation of the application. The other process, the system server, serves the requests for synchronization and requests for the most recent updates of a page of shared memory. We will just briefly describe our implementation of the synchronization objects, and not go into the details of our implementation of shared memory.

Each synchronization object resides on one of the system servers. Every synchronization object has a unique id number and each method in the public section of the synchronization class is assigned an id number. When a thread accesses a method of a synchronization object, the thread itself sends a message to its local system server about which object and operation it is accessing. The local system server processes the request if the synchronization object resides locally. Otherwise it forwards the request to the remote system server where the object resides.

SyncClass'es are written by the programmer. A preprocessor generates two classes by parsing it, one is for threads and the other is for system servers. The class for threads contains only the methods declared in the public section of the synchronization classes. The function of these methods is to compose a message to the system server and wait for the response from the server if necessary.

The generated class for system servers has the actual code for the user-defined methods of the synchronization class, plus some extra operations for communicating the results and the information for updating shared memory.

5 Performance

The platform we used to evaluate performance are 8 PCs with Pentium Pro processors and 64 Megabytes RAM connected by by 100Mbps Ethernet. Reported times are wall clock times.

We present performance results for computations of the Mandelbrot set. We chose the Mandelbrot computation because it is relatively easy to understand and illustrates the use of synchronization objects well. The algorithm used to compute the Mandelbrot set is taken from [GLS94], and will be briefly described here.

The protocol we use to implement the memory model is called single owner protocol[CK96]. Single owner protocol is also a multiple writer protocol[CBZ95, KCDZ94]. For each page on shared memory there is a page owner which keeps the current version of the page and serves read request from other threads. We also run the program without high level synchronization primitives using the lazy invalidation protocol for comparison.

5.1 Application

The Mandelbrot set is defined as the set of all complex numbers c that satisfy the following criteria.

Define the function $f_c^n(z)$ as the repeated application of $f_c(z) = z^2 + c$, where c and z are numbers on the complex plane. Thus $f_c^1(z) = f_c(z)$ and $f_c^2(z) = f_c(f_c(z))$. If $f_c^n(0)$ stays bounded as $n \to \infty$, then we say that c is a member of the Mandelbrot set.

It can be shown that if $|f_c^n(0)| > 2$ for some n, then f_c is not bounded at c. For computational purposes, this means we can pick a suitably large number N and calculate $f_c^n(0)$ for $n < N$. If f_c stays bounded for N steps, then we give up and add it to the set. If not, then c is not in the set.

We can represent each complex number c as a pixel of an image. If f_c stays bounded for N steps we can give it some color, say black. If f_c becomes unbounded after n steps, we can give it another color that is some function of n. Thus, a sequential version of this program can iterate through all points of the image, assigning a color to each pixel. In this way, a striking picture develops.

An optimization to the sequential algorithm is described in [GLS94]. The authors note that if the border of any square of pixels are all the same color, then the interior must be of the same color also. Thus our new algorithm starts by checking the boundary of the region, and if it is all the same color, it colors the interior. If not, it breaks the region into equal sized blocks and recursively calls itself on each new region. If a particular block is of a minimum size, it gives up and colors the interior anyway sequentially. This substantially speeds up the computation.

5.2 Implementation

The simple parallelization of this algorithm involves putting the blocks in a shared task pool. Each worker accesses the pool, computes the block and does one of two things: it either breaks up the block and puts the new pieces in the shared task pool or it colors the whole block (either because it is of minimum size or because it is all one color).

Because the computation time of each block is small compared to the cost of communication on a typical network of workstations, we adopt a modification of the original scheme to make it slightly coarser grained.

Each worker will have its own local task pool from which it grabs the blocks. Periodically, say after doing N blocks, the worker will consult with the global pool to see whether it should take or add blocks to the global pool to more properly balance the workload. The way it decides this is based on an approximation of the total number of existing blocks. It takes this number and divides it by the number of workers. This result can be considered the number of blocks each processor should be working on (in the ideal case). If the current worker has more than that number of blocks, it takes some from its local pool and adds it to the global pool. If it has less, it takes from the global and adds to the local. If the worker cannot get any block to work on, it sleeps and waits until there are available blocks in the pool. In our experiment, we did not include the time to display the result on the screen since it takes much longer than the time to compute the colors of all pixels on the screen.

We show the results for two programs. The first program uses a user defined high level synchronization primitive, *PoolController*, to keep the control information. There are two operations used to synchronize workers, *GetInfomation* and *Done*. *GetInformation* is annotated with a *get* attribute. It also returns the information about what the worker should do next — i.e. whether it should take or add blocks to the global pool. Then the worker can either get or put some blocks into the global pool or it may not need to access the global pool at all and can just work on those blocks on its local pool. The worker can also be informed that the computation is finished. *Done* is annotated with a *put* attribute. After a worker finishes accessing the global pool, it executes *Done* to inform *PoolController*.

When the *PoolController* receives a request for *GetInfomation* from a worker, the server updates the information about the status of the worker and sends how many jobs the worker needs to do next. If the worker needs to access the global pool, the server does not respond to any other worker until the worker performs the *Done* operation. If there is no available blocks on the global pool for the worker, the server postpones responding to the worker until some other worker puts blocks on the global pool. If all workers are waiting for blocks, the server sends a terminating response back. Then all workers finish the computation.

The second program uses semaphores to synchronize the workers. Several semaphores are needed to emulate a high-level synchronization object. Decisions about the number of semaphores needed and coordination amongst them require careful coding from the programmer. We use two semaphores, called *mutex* and *sleep*, to synchronize workers. The control information about status of other workers, how many blocks are on the global pool, how many workers sleep and wait for blocks, and the pool itself, is protected by *mutex*, a binary semaphore. Every time a worker accesses the control information it needs to execute the P operation of *mutex*. Then it obtains the control information and decides to either write or read the global pool. Afterwards, it executes the V operation of *mutex* when it leaves the critical section. If the worker finds there is nothing to work on, it goes to sleep by executing the P operation of *sleep*. It also may wake up other sleeping workers by executing the V operation of *sleep*.

5.3 Performance Result

We checked the Mandelbrot set on the plane where $x = [-2, -1.25]$ and $y = [0.5, 1.25]$. Assume there are 720×480 points on the plane. Each point on the plane is tested by the iteration function, $f_c^n(z)$ defined at section 5.1 with $z = 0$. If the function stays bounded after 256 iterations, we consider c is in Mandelbrot set.

Sequential execution time is 18.2 seconds. The execution time of single owner protocol, with high level synchronization primitives, on eight processors is 3.72, see figure 3. Its speedup is 4.91. The execution time of single owner protocol without high level synchronization primitives is 5.8 seconds on eight processors. Its speedup is 3.2. The execution time of lazy invalidation protocol without

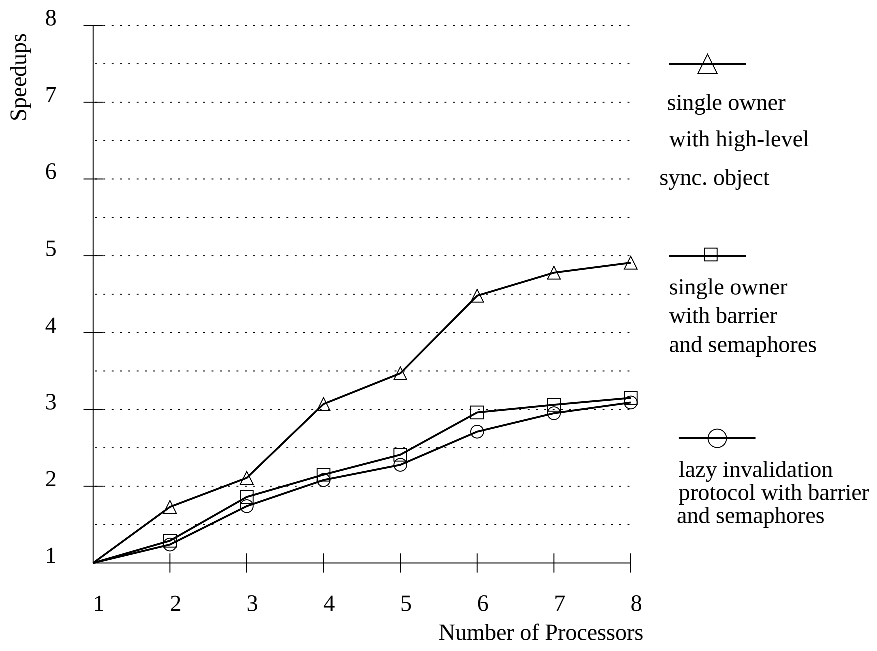

Figure 3: Speedup for Computing Mandelbrot Set

high level synchronization primitives is 5.9 seconds on eight processors. Its speedup is 3.1.

Table 4 shows the program with the high level synchronization primitive uses fewer messages and the message size is also smaller than the program with basic synchronization primitives. The high level synchronization primitive itself eliminates fine grained access to control information for load balancing and the status of the global pool. This information is stored in the server locally. Putting a worker to sleep is done by postponing responding to the worker. In contrast, with semaphores, the worker goes to sleep by calling another semaphore, resulting in more messages.

6 Related Work

put and *get* operations are similar to *release* and *acquire* in release consistency [GLL$^+$90]. In contrast, *put* and *get* operations are performed with respect to synchronization objects but *release* and *acquire* are performed with respect to other threads. None of these systems have user-definable synchronization primitives, but only provide basic locks and barriers.

In entry consistency[BZS93], synchronization objects are also restricted to either locks or barriers. Entry consistency requires an association between shared data and its guarding synchronization object S. The work of association is done by the programmer. This association is used to piggy-back

number of processors		2	3	4	5	6	7	8
single owner protocol with high level sync.	number of messages	103	133	204	257	291	311	375
	sizeof messages (K bytes)	5.2	8.3	14.2	19.2	23.1	27.7	35.7
single owner protocol with basic sync	number of messages	188	254	423	476	593	614	780
	sizeof messages (K bytes)	11.3	19.6	37.6	65.8	90.7	97.7	139.2
lazy inv-alidation protocol with basic sync.	number of messages	215	313	412	554	648	719	760
	sizeof messages (K bytes)	10.1	20.8	40.2	67.4	98.5	138.1	163.5

Figure 4: Speedup for Computing Mandelbrot Set

fine-grained data with the acquire and release operations. The updates of the guarded shared data of thread t are visible to other threads after t performs a release access on S. Thread t's updates are not visible to thread t' until t' performs an *acquire* access on S. Only those variables associated with S are visible to thread t'. The execution between acquire access and release access in the entry consistency memory model is mutually exclusive and called a critical section. A thread can not access the critical section until the previous acquiring thread performs a release access.

In contrast, with Smile consistency, programmers do not need to associate shared datum with a synchronization object. All writes on shared memory are visible to the subsequent *get*'ting thread. However, the fine-grained data can be still passed around by synchronization objects, by the parameters and return values of the user-defined methods. Execution between a *get* operation and a *put* does not have to be mutual exclusive. It is up to the definition of synchronization class.

Like Calypso[BDK95] and Cilk[BJK+95], we concentrate on giving the user a more friendly interface to shared memory and do not emulate sequential consistency. But synchronization is done through the spawning and joining of threads in Calypso and Cilk. Our memory model is similar to Cilk's DAG (directed acyclic graph) consistency, but their DAG is not built through operations on a synchronization object. Instead the DAG is based on thread creation and completion. Cilk and Calypso's focus is on fault tolerance and adaptive load balancing.

7 conclusion

We present a program centric memory model, Smile. Smile provides a user friendly memory model and a paradigm for users to compose their own synchronization primitives. We also show a program with high level synchronization primitives may outperform a program with basic synchronization primitives.

Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grant number CCR-94-11590; and by the Intel Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[ACD+96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Peter Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[BDK95] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *proceeding ofthe 4th IEEE Intl. Symp. on High Performance Distributed Computing*, 1995.

[BJK+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson abd K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Pratice of Parallel Programming(PPoPP)*, 1995.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. Technical Report CMU-CS-93-119, Carnegie-Mellon University, 1993.

[CBZ95] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[CK96] Churngwei Chu and Zvi Kedem. Techniques for improving the performance of multiple writer memory protocols in distributed shared memory systems. In *Proceedings of High Performance Computing*, pages 260–265, 1996.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *the Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *USING MPI:Portable Parallel Programming with the Mesage-Passing Interface*. The MIT Press, 1994.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *The 1994 Winter USENIX Conference*, 1994.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*, chapter 2:Processes, pages 27–70. Prentice Hall, 1992.