Distributed Cactus Stacks: Runtime Stack-Sharing Support for Distributed Parallel Programs[1]

Shantanu Sardesai [A], *Donald McLaughlin* [B] *and Partha Dasgupta* [B]

[A] Tandem Computers Inc.
19333 Vallco Parkway,
Cupertino, CA 95014-2599
shantanu.sardesai@tandem.com

[B] Department of Computer Science and Engineering.
Arizona State University
Tempe AZ 85287-5406
partha@asu.edu

Abstract

Parallel Programming Systems based on the Distributed Shared Memory technique has been promoted as easy to program, natural and equivalent to multiprocessor systems. However, most programmers find this is not the case. The shared memory in DSM systems do not have the same access and sharing semantics as shared memory in real multiprocessor systems (shared memory multiprocessors). We present a scheme, which has been implemented as a part of the Chime Parallel Processing system, that provides, true shared-memory multiprocessor semantics in a distributed system. Programs are written as parallel programs with constructs for parallel for-loop and parallel compound statement. A runtime system (middleware) provides the above features using a unique multithreaded architecture. In addition to providing the stack sharing support, the runtime system is able to provide nested parallelism, task synchronization, load balancing and fault tolerance. The software is available at http://milan.eas.asu.edu.

Keywords: *Parallel Processing, Workstation Clusters.*

1. Introduction

Shared memory multiprocessors are regarded as the most "programmer-friendly" platform for writing parallel programs. These platforms support a variety of parallel processing languages that provide constructs for expressing shared data, parallelism, synchronization and so on. However, the cost and lack of scalability and upgradability of shared memory multiprocessor machines, make them a less than perfect platform. Distributed Shared Memory (DSM) has been promoted as the solution that makes a network of computers look like a shared memory machine. This approach is supposedly more natural than the message passing method used in PVM [GBD+94] and MPI [GLS94].

However, most programmers find this is not the case. The shared memory in DSM systems do not have the same access and sharing semantics as shared memory on shared memory multiprocessors. For example, only a designated part of the process address space is shared, linguistic notions of global and local variables do not work intuitively, parallel functions cannot be nested and so on.

Chime is the *first* system that provides a true shared memory multiprocessor environment on a network of machines. It achieves this by implementing the CC++ [CK92] language (shared memory part) on a distributed system. In addition to shared memory, parallelism and synchronization features of CC++, Chime also provides fault-tolerance and load balancing [Sar97, Das97].

2. Related Work

Shared memory parallel processing in distributed systems is limited to a handful of DSM systems

[1] This research is partially supported by grants from DARPA/Rome Labs, Intel Corporation and NSF.

that provide quite similar functions (Midway [BZS93], Quarks [Kha96], TreadMarks [ACD+96], Munin [Car95]). DSM systems do not provide a uniform view of memory i.e. some global memory is shared, but not the other. In addition, the parallel tasks execute in an isolated context; i.e. they do not have access to variables defined in the parent's context. Moreover, a parallel task cannot call a function that has an embedded parallel step (nesting of parallelism is not allowed).

The *Calypso* system [BDK95, Cal96, MSD97] adds fault tolerance and load balancing to the DSM concept, but suffers from the lack of nesting and the lack of synchronization (except barrier synchronization).

Other systems provide *hybrid* programming environments. A mixture of constructs is used for expressing serial execution and parallel execution in Linda [CG89] and GLU [JF92]. CC++ is a hybrid programming language that supports two kinds of programming paradigm—the shared memory paradigm and the distributed memory paradigm. The same program can use both constructs there by making design development and debugging difficult. The shared memory constructs work on multiprocessors and the distributed memory constructs work on the network. In our research, we discovered techniques to make the shared memory constructs of CC++ work on a network.

3. Chime

In this paper we describe the runtime system support in general and the support for *distributed cactus stack* in specific, in the implementation of the Chime parallel processing system. Chime is a shared-memory based distributed parallel processing system that provides:

- *True multi-processor shared-memory semantics on a network of workstations*: It provides programmers with single uniform view of memory and hides the details of data partitioning and data distribution.
- *Block structured scoping of variables and non-isolated distributed parallel execution*: The block structured scoping of variables is provided by implementing a *distributed cactus stack*. Unlike most "distributed" parallel processing systems, the parallel statements start their execution from within the context of the function that executes them.
- *Support for nested parallelism*. The nesting can either be linguistic (i.e. a parallel construct contains a nested parallel construct) or at runtime (i.e. a statement inside a parallel construct calls a function that has an embedded parallel construct).
- A large degree of fault isolation and load balancing, with no programmer intervention. All but one of the machines participating in a computation can fail and possibly recover at any time without affecting the correctness of the computation. Chime also automatically distributes the workload among the available machines such that faster machines do more work compared to slower machines. Unlike most other fault-tolerant systems, there is no *additional cost* associated with these features in the absence of failures. (The implementation of fault-tolerance and load balancing is beyond the scope of this paper).
- Inter-task synchronization is supported using single assignment variables. (The implementation of synchronization mechanism is beyond the scope of this paper).

Chime provides a programming interface that is based on the *Compositional C++* or CC++ [CK92] language definition. In Chime, a programmer develops applications for a shared memory multiprocessor model, which are then pre-processed into C++ programs. Consider the following parallel program:

```
#include <iostream.h>
#include "chime.h"
    #define N 1024
int GlobalArray[N];

void AssignArray(int from, to){
  if (from != to)
    par{
      AssignArray(from, (from+to)/2);
      AssignArray((from+to)/2+1, to);
    }
  else GlobalArray[from] = 0;
}
int main(int argc, char *argv[]) {
       AssignArray (0, N-1)
}
```

The two statements in the block are executed in parallel on remote machines

The above program defines a global (*shared*) array called `GlobalArray`, containing 1024

integers. Then it assigns the global array using a recursive parallel function called `AssignArray`. The `AssignArray` function uses a "`par`" statement. The `par` statement executes the list of statements within its scope in parallel, thus calling two instances of AssignArray in parallel. Each instance calls two more instances, and this recursion stops when 1024 leaf instances are running.

The above program (and ones that are more complex) runs without any modifications, on a network of workstations under the Chime Parallel Processing system. Other DSM systems cannot execute the above program for some of the following reasons:

- The parallel tasks start their execution from within the context of the parent task.
- Siblings of the task share the global variables, and any local variables defined before the scope of a parallel task starts.
- The program has a variable degree of parallelism, due to its nesting of parallel statements.

Chime can execute the above program by first pre-processing it to C++, compiling and linking the resulting program with the Chime runtime library. The compiled programs are executed by the runtime system, on a virtual machine, implemented on a distributed network. This approach not only provides the programmer with a completely transparent distributed shared memory combined with traditional block structured programming, but it also allows transparent utilization of unreliable shared resources available over the network. The resource utilization and handling of unreliable computing resources is provided by dynamic load balancing and fault tolerance features. Measurements show that there is almost no additional overhead for these features [MSD97, Sar97]. Chime has been implemented in C++, using Microsoft Visual C++ 4.2 and was developed on Pentium-based machines running Windows NT 4.0.

4. Distributed Cactus Stacks

When different nested parallel tasks start their execution from within the context of their parent task on different workstations, they create a *Distributed Cactus Stack*. A cactus stack conceptually, grows from the initial parent stack and bifurcates into multiple different branches, which vary depending upon the control flow of the parallel tasks. All the parallel tasks share the initial common stack but have their own, branched stack after the beginning of the parallel task.

Each parallel task inherits its parent task's stack and grows it further based on its own execution, which may be different from its sibling parallel tasks. All the siblings that have same parent task share the activation record for their parent task and can perform updates. Chime runtime system implements the distributed cactus stack and provides sharing of the above data transparently. This enables the programmer to use traditional block structured scoping of variables to exploit nested parallelism in a multi-processor like shared memory, parallel processing environment on a network of workstation.

Consider the following program fragment:

```
01  foo ( . . . ) {
02  int x = 10;
03  parfor (int index=1;
            index <=5, index++)
04     incNprint(x,index);
05  }
06
07  incNprint(int &x, int z){
08  int y = 0;
09     par {
10       y = x + z;
11       cout << y << endl;
12     }
13  }
```

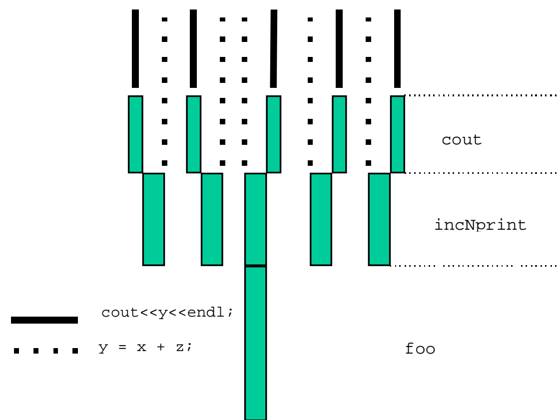
The **`par`** and **`parfor`** statements are keywords in CC++. The `parfor` statement allows data-parallel executions by running each iteration of the loop as a separate concurrent thread. The `par` statement allows task parallel executions by running each statement in a `par` block as a separate thread.

In the above program, the variable `x` is defined in line 02, and hence it is shared by all tasks running in the `foo` routine, *even though it is not a globally declared variable*. It must be noted that the variable is accessible to parallel tasks running on remote machines. The variable `y` defined in line 08 is shared by the pair of sibling `par` tasks, which are children of one of the tasks running a particular index value (of the `parfor` statement). Each task in the `parfor` statement has a private,

pre-initialized copy of the index, but the index is shared by all the subtasks of each `parfor` task.

Thus, the above program generates 10 tasks, in addition to the parent. Moreover, 5 integers are printed, some values are 0 (when the print runs before the assignment) and some are the numbers from 11 to 15. Note: this program actually violates the memory usage guidelines of CC++ and hence the execution results are implementation dependent.

The program above, at execution time, builds a stack, which conceptually looks like the figure below:



As stated earlier, the Chime runtime system implements the distributed cactus stack transparently for the application. When a parallel task starts its execution on a remote workstation the runtime system gets the stack of the parent task and sets up the stack for this task. The runtime system is also responsible for maintaining the *distributed cactus stack* as it grows and shrinks. After completion of the parallel tasks, the runtime system collects all the updates transparently. It also sends all the stack updates back to the parent task once the execution of the parallel sub-task is complete.

The semantics of accessing shared variables in Chime is the same as that in CC++, i.e. concurrent read exclusive write (CREW). Thus, any two parallel tasks can read value from a same location concurrently but if a parallel task writes to location no *other* parallel task should read value from or write to the location. The granularity of memory updates is a byte.

5. Chime Architecture

A program written in CC+ is preprocessed to convert it to C++ and compiled. Then it is linked with the Chime runtime library and a single executable file is generated. This executable is executed on a network of machines (or workstations). One of the workstations is designated as the *manager* and the rest as *workers*. All run the same executable.

A program starts on the manager. When the program reaches a parallel construct, parallel tasks are generated and are allocated by the manager to the waiting workers. During the execution of the parallel step, the manager does scheduling and allocation of parallel tasks, as well as memory management.

Workers pick up task assignments from the manager. Then they execute the tasks and notify the manager on termination of the task. Workers access shared memory using page faults, as in any page-based DSM system. The manager services all requests for reading and writing memory.

The manager process consists of two threads, called the *application thread* and the *control thread*. The application thread executes the code programmed by the programmer. The control thread executes, exclusively, the code provided by the Chime library. Hence, the application thread runs the program and the control thread runs the management routines, such as scheduling, memory service, stack management, synchronization handling and so on.

Similarly, the worker process also consists of two threads - the *application* thread and the *control* thread. The application thread in the worker and manager are identical. However, the control threads are not. The control thread in the worker is the client of the control thread in the manager; i.e. it requests work from the manager and retrieves data pages from the manager.

5.1 Preprocessing Parallel Statements

Consider the following parallel statement:

```
parfor ( int i=0; i<100; i++) {
      a[i] = 0;
};
```

The above statement creates 100 tasks, each task assigning one element of the array **a**. The array **a** is defined somewhere within the scope of the **parfor** statement (could be global, or could be local).
The preprocessor converts the above statement to something along the following lines:

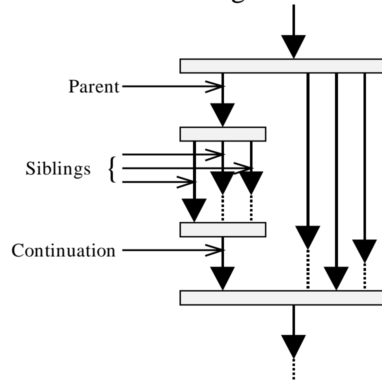
```
1.  for ( int i=0; i<100;i++){
2.      add task entry and &i
        in the scheduling table;
    }
3.  SaveContext of this thread;
4.  if worker { {
        a[i] = 0;
        };
5.      terminate task;
    }
6.  else
7.      suspend this thread and
        request manager to
        schedule threads till
        all tasks completed;
```

The above code may execute in the manager (top level parallelism) or the worker (nested parallelism). *Assume the above code executes in the manager*. Then the application thread of the manager executes the code. Lines 1 and 2 create 100 entries in the scheduling table, one per parallel task. Then line 3 saves the context of the parent task, including the parent stack. Then the parent moves to line 7 and this causes the application thread to transfer control to the control thread.

The control thread at this point waits for task assignment requests from the control thread of workers. When a worker requests a task, the manager control thread sends the stored context and the index value of **i** for a particular task to the worker.
The control thread in the worker installs the received context and the stack on the application thread in the worker and resumes the application thread. *This thread now starts executing at line 4.* Note that now the worker is executing at line 4, and hence does one iteration of the loop and terminates. Upon termination, the worker control thread regains control, flushes the updated memory to the manager and asks the manger for a new assignment.

5.2 Scheduling

The controlling thread at the manager is also responsible for task assignment, or scheduling. The manager uses a scheduling algorithm that takes care of task allocation to the workers as well as scheduling of nested parallel tasks in correct order. Nested parallel tasks in an application form a DAG as shown in the figure below.



A DAG for a nested parallel step.

Each nested parallel step consists of several parallel tasks, which are called *siblings*. It also has a *parent* task and a *continuation* that must be executed, once the nested parallel step has been completed. A continuation is an object that fully describes a future computation. It represents the remaining execution of the parent parallel task that must be resumed after the nested parallel step has been completed. To complicate the scenario, a continuation may itself have nested parallel step(s).

The manager maintains an *execution dependency graph* to capture the dependencies between the parallel tasks and schedules them and their corresponding continuations in correct order. The allocation of tasks to the workers is done by *eager-scheduling* [DKR95, BDK95] which replicates computations in a non-conventional fashion whenever, more than required computational resources are available. To ensure correctness in eager scheduled computations, the execution uses TIES (two-phase idempotent execution strategy) [BDK95] for memory management. This has two benefits: First, it provides fault tolerance for free as any worker may fail during the execution of a parallel task, without affecting the computation at all. Second, it also provides load-balancing for free as fast

workers do not wait for slow workers and do more work. Actually, this also speeds up the overall computation as fast workers do not wait for slow workers to finish the work. And the coupling of eager scheduling with TIES provides a very efficient distributed execution scheme [BDK95, MSD97] [2]

5.3 Managing Cactus Stacks

The management of cactus stacks happens in an integrated fashion, along with the task scheduling and task instantiation methods outlined above. For top level nesting, the manager process is suspended at a point in execution where its stack and context should be inherited by all the children threads. When a worker starts up, the worker is sent the contents of the manager's stack along with the context (i.e. register values, program counter and so on). The controlling thread of the worker process then installs this context as well as the stack, and starts the application thread, which of course finds itself automatically executing at the statement the manager, was suspended. After the thread is completed, the updates are sent back and applied to the manager's global state and the manager's stack.

However, if a worker executes a *nested* parallel step, the same code as the above case is used, but the runtime system behaves slightly differently. The worker, after generating the nested parallel jobs, invokes a routine that adds the jobs and the continuation of the parent job to the manager's job table, remotely. The worker suspends and the controlling thread in the worker, sends the worker's complete context, including the newly grown stack, to the manager. Thus, when the manager creates the subtasks on more workers, it sends the state of the worker that created the threads. This leads to the proper growth of the cactus stack.

The stack for a nested parallel task, therefore, is constructed by writing the stack segments of its ancestors onto the stack of a worker's application thread. In order to reconstruct this stack successfully, the task table entry of each nested parallel task must also contain a set of stack pointers that represent the location where these stack segments must be placed.

Upon completion, the local portion of the stack for a nested parallel task is unwound leaving only those portions that represent its ancestors. This portion of the stack is then XOR'ed with its unmodified shadow and the result, which contains only modifications, is returned to the manager. In the manager, the stack pointers associated with the task are then again examined to determine which portions of the returned stack are associated with each ancestor. These portions are then XOR'ed with the stack segments that are associated with each of nested parallel task's ancestors.

6. Performance

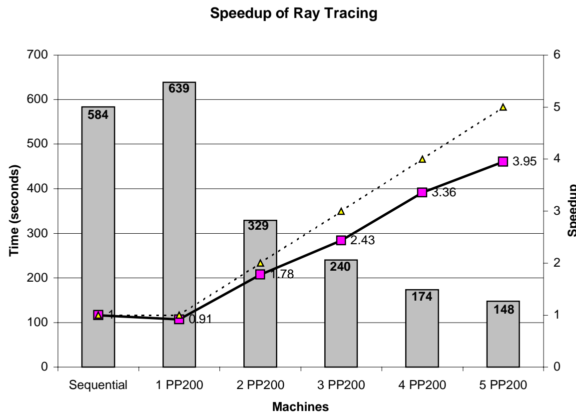
Many performance tests have been done on Chime[Sar97], including its prowess in speedups, load balancing and failure management. Here we present the speedup tests and the overhead measurements of the cactus-stack management routines.

The performance tests used five Pentium Pro 200MHz (P-200) machines, with 32Mbytes of memory, running Windows NT 4.0 connected via a 100Mbit/sec Ethernet. The timings in all tests were done with a real, physical stopwatch and not system times, or user times or any such measures. Thus they are real "*wallclock*" times.

6.1 Speedup Test

A Ray Tracing program was first written as a sequential version, that is a real proper sequential program with no Chime embellishments. The parallel Ray Trace Chime program was written with two parallel steps and 40 tasks per parallel step). The program was used for the "*speedup test*" and the sequential execution was used as the base case.

[2] Eager Scheduling and TIES form the basis of our fault-tolerance and load balancing mechanism. Complete description of these mechanisms is beyond the scope of this paper.

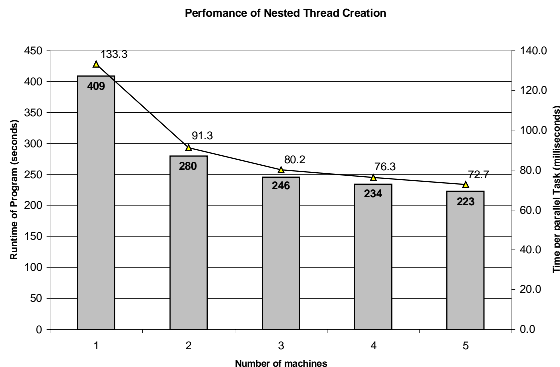


The parallel program was run on one P-200 with manager and worker running on the same machine. The execution took 639 seconds. The extra 55 seconds accounted for all overhead (networking, memory copying, recreating context, scheduling and so on.). The figure above shows the performance results for a maximum of five P-200 machines.

Note that the speedups obtained are competitive to other parallel processing systems such as PVM, TreadMarks and Quarks. This is in spite of Chime providing support for shared memory, distributed cactus stack, non-isolated remote execution, fault tolerance and load balancing.

6.2 Cactus-Stack Overhead

Now we present the cactus-stack overhead measurements of the management routines. To test the performance we used the `AssignArray` routine shown in Section 2 of this paper. We ran this routine on a varying set of machines and observed the total runtime of the program.



As can be seen from the program code, a 1024 element array is initialized, by using 1024 parallel tasks at the leaf level. Thus there are a total of 3068 tasks that are created and destroyed. For each task, a remote execution is started, a context and stack is loaded and then the task is terminated by going though the code for sending back updates (global updates happen in this case).

Thus, a total time of 409 seconds (133ms/task) is spent if a one worker and one manager are run on one single machine. As we add more machines, the time taken drops to about 230 seconds (about 74ms/task). This is about the lowest we reach, as this is the point where the central manager saturates at creating jobs, transmitting and receiving data and performing management chores. Note that the saturation point is only five workers, as each worker is doing no work, but requiring constant manager attention.

Given an overhead of about 70ms per task creation, including all overheads, including some rudimentary data transfer, we feel is a good performance number and is competitive to other DSM system that do not support much of the features of Chime. In addition, other tests of speedup and fault-tolerance also show respectable performance qualities of Chime.

7. Conclusions

The use of a distributed cactus stack in Chime allows us to provide runitme support for multiprocessor languages on a distributed system. While Chime is built around the CC++ langauge, these techniques are applicable to any langauge supporting block-structure, shared memory and parallel statements.

The performance results show that adding scoping of variables via the cactus stack implementation is feasible from a performance standpoint, in a distributed system..

8. References

[ACD+95] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer,* December 1995.

[AS91] Brian Anderson and Dennis Shasha. Persistent Linda: Linda + Transactions + Query Processing. *Workshop on Research Directions in High-Level Parallel Programming Languages*, Mont Saint-Michel, France June 1991.

[BBD+87] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[BCZ90] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. 2nd Annual Symp. on Principles and Practice of Parallel Programming*, Seattle, WA (USA), 1990. ACM SIGPLAN.

[BDK95] A. Baratloo, P. Dasgupta, and Z. M. Kedem. A Novel Software System for Fault Tolerant Parallel Processing on Distributed Platforms. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing, 1995.*

[BJ87] K. P. Birman, and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions of Computer Systems*, Vol. 5, no. 1, pp. 47-76.

[BZS93] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of COMPCON '93*, February 1993, pp528-537.

[Cal96] The Milan, Chime and Calypso Home Pages. http://milan.eas.asu.edu

[Car95] J. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, 29: pages 210, 227, 1995.

[CG89] N. Carriero and D.Gelernter. Linda in Context. Comm. of ACM, 32, 1989.

[CK92] K. M. Chandy and C. Kesselman, *CC++: A Declarative Concurrent, Object Oriented Programming Notation*, Technical Report, CS-92-01, California Institute of Technology, 1992.

[Das97] P. Dasgupta, *Parallel Processing with Windows NT Networks*, The USENIX Windows NT Workshop, 1997.

[DKR95] P. Dasgupta, Z. M. Kedem, and M. O. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems, 1995.*

[GBD+94] Al. Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, 1994.

[GLS94] W. Gropp, E. Lusk, A. Skjellum. Using MPI Portable Parallel Programming with the Message Passing Interface. *MIT Press, 1994*, ISBN 0-262-57104-8.

[JF92] R. Jagannathan and A. A. Faustini. GLU: A Hybrid Language for Parallel Applications Programming. Technical Report SRI-CSL-92-13, SRI International. 1992.

[Kha96] Dilip R. Khandekar. *Quarks: Distributed Shared Memory as a Basic Building Block for Complex Parallel and Distributed Systems.* Master's Thesis. University of Utah. March 1996.

[MSD97] D. Mclaughlin, S. Sardesai, and P. Dasgupta. *Calypso NT: Reliable, Efficient Parallel Processing on Windows NT Networks*, Technical Report, TR-97-001, Department of Computer Science and Engineering, Arizona State University, 1997.

[Ric95] Jeffery Richter, *Advanced Windows: The Developers Guide to the Win32 API for Widows NT 3.5 and Windows 95* , Microsoft Press, Redmond, WA, 1995

[Sar97] Shantanu Sardesai, *CHIME: A Versatile Distributed Parallel Processing System*, Doctoral Dissertation, Arizona State University, Tempe, May 1997.

[SpBe97] E. Speight and J. K. Bennett, *Brazos: A Third Generation DSM System*, The USENIX Windows NT Workshop, 1997.

[Sun90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315-339, 1990.

Sponsor Acknowledgment: Effort sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.
Sponsor Disclaimer: 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.