Distributed Shared Array: An Integration of Message Passing and Multi-Threading on SMP Clusters

Cheng-Zhong Xu, Brian Wims and Ramzi Jamal
Department of Electrical and Computer Engineering
Wayne State University, Detroit, MI 48202
czxu@ece.eng.wayne.edu

Abstract

This paper presents a Distributed Shared Array runtime support system to support Java multithreaded programming on clusters of SMPs. The DSA programming model exposes the cluster's hierarchical organization to programmers and allows them to explicitly control data distribution while relieving them from orchestration of inter-node communication at run-time. The DSA system was developed as a mobile agent and the DSA-based virtual machine could be reconfigured to adapt to the varying resource supplies or demand over the course of a computation.We demonstrated the programmability of the model in parallel sorting and LU factorization problems and evaluated their performance on a cluster of SMP servers.

1 Introduction

With the advent of low-latency, high bandwidth interconnection networks and the popularity of symmetric multiprocessors, clusters of SMPs (CLUMP) are emerging as a cost-effective way for high performance computing. The CLUMP architecture offers the promise of scalability and cost-effectiveness. However, delivering its full potential heavily relies on an easy-to-use and efficient programming environment that overlays the nodal operating systems. Two primary parallel programming models are message-passing and shared-address-space. The message-passing model, embodied in PVM/MPI-like libraries, tends to deliver high performance on distributed memory MPPs and clusters of workstations. However, it is inadequate on CLUMPs due to its flat view of memory. The message-passing model is also proven hard to program because programmers are required to explicitly distribute data across processes' disjoint address spaces and schedule inter-process communications at run-time. There are software-based distributed shared memory (DSM) systems, such as Ivy [12] and TreadMark [1], in support of the shared address space model on clusters. They simplified parallel programming with a compromise of efficiency. Since a CLUMP employs a deeper memory hierarchical organization, it is the depth of the hierarchy and the non-uniform access costs at each level that make the construction of an efficient DSM system on CLUMPs even harder.

A recent research trend is to develop multithreaded DSM systems to explore the potential of SMP nodes. Examples include Brazos [16], CVM [11], and String [15]. They improved upon the early DSM systems by adding threads to exploit data locality within a SMP node. Between the objectives of ease-of-programming and efficiency, they still bias the first and their actual performance is yet to be seen. This paper presents a Distributed Shared Array (DSA) runtime support system for a tradeoff between the two objectives in another way. It supports Java-compliant SPMD multithreaded programming and exposes to programmers the cluster's hierarchical organization. It allows the programmers to control data distribution while relieving them from runtime scheduling of inter-node communications.

The DSA system shares similar objectives with Global Array [14] and Split-C [5] to combine the better features of message passing and share address spacing for a fairly large class of regular applications. It joins a number of recent projects towards Java-based parallel programming environments. JPVM [8] and mpiJava [2] provides interfaces of the PVM and MPI libraries to Java so as to support high-level message-passing programming in Java. Charottle [3][10] provided object-based shared memory for objects of certain distributed classes. Java/DSM [18] suggested modifying Java Virtual Machine on top on existing DSMs. Unlike the existing parallel programming environments, the DSA system has the following unique features.

- It is tailored to CLUMPs with a disclosure of nodal architecture to the programmers. Programmers are responsible for thread creation and assignment. It supports data replication to tolerate remote access latency and provides a mobile ownership mechanism to ensure data coherence.
- It is developed as an integral part of a mobile agent based computing infrastructure, TRAVELER, to support users' multithreaded computational agents. The DSA system itself is implemented as a mobile agent so that the DSA-based virtual machine can be reconfigured or migrated to adapt to the change of resource supplies or requests.

The rest of the paper is organized as follows. Section 2 presents the DSA programming model. Section 3 shows the implementation details of the DSA runtime support system. Section 4 presents preliminary experimental results. Section 5 summarizes the results with remarks on future work.

2 Overview of the DSA

The DSA system provides a Java-compliant programming interface to extend multithreaded programming to clusters. It supports single-program multiple-data (SPMD) programming paradigm. It exposes the hierarchical organization to programmers and allows programmers to explicitly specify globally shared arrays and their distributions. Shared arrays are distributed between threads, either regularly or as the Cartesian product of irregular distributions on each axis.

The DSA system relieves programmers from run-time scheduling of inter-node communications. Remote data access is supported by a run-time system, as shown in Figure 1. The DSA runtime system refers a node to the machine that has one or more processors. It consists of a group of DsaAgents running at each node and together forming a distributed virtual machine. The DsaAgents are responsible for local and remote access to shared objects.

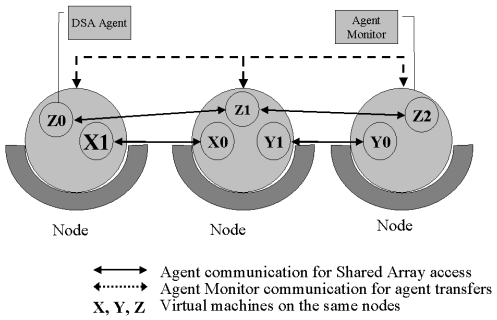

Figure 1. Architecture of the DSA

DsaAgents are run under the watch of an AgentMonitor. Each node has a residing AgentMonitor, which monitors the execution of local DsaAgents and coordinates with other AgentMontiors to migrate DsaAgents for load balancing and fault tolerance. Following is the procedure for a node to create a DsaAgent and associate it with an AgentMonitor.

```
mon = new AgentMonitor( );
Naming.rebind("Monitor", (AgentMonitor)mon);
dsa=new DsaAgent(mon, amtThreads);
        // programmers need to specify the number of threads that are
        in proxy by the DsaAgent.
dsa.initDsaAgent(mon)
```

3 DSA Programming Model

The DSA programming model relies on two core objects: DsaAgent and SharedArray. In the following, we present details of the objects and illustrate their usage via an example.

3.1 DsaAgent and Shared Arrays

DsaAgent implements an Agent interface. An agent is a special object type that has autonomy. It behaves like a human agent, working for some clients in pursuit of its own agenda. Details of the Agent interface will be discussed in Section 4.2. Due to the hierarchical organization of SMP clusters, we refer to properties related to individual nodes as "local". We designate the thread-0 of node-0 as the main thread.

```
public class DsaAgent implements Agent {
        int setNodeId(), getNodeId();
        int numGlobalThreads(),  numLocalThreads();
        int localThreadId(), globalTheadId();

        SharedArray createSharedArray (String  name, int size, int grain)
        SharedPmtVar createPmtVar (String name)
        void globalBarrier(), localBarrier()
        Barrier createBarrier(String name, int size, String type)
}
```

The DSA system defines two distributed variables: SharedArray and SharedPmtVar. The method createSharedArray( ) creates a SharedArray object. It can be distributed between threads in different ways. Currently, block decomposition is supported. The parameter grain specifies the granularity of coherence for data replication. Since Java doesn't allow operator overloading, the DSA actually provides three extensions of SharedArrays: SharedIntArray, SharedFloatArray, and SharedDoubleArray. Similarly, the object SharedPmtVar refers to a base of shared objects of primitive type. The method createPmtVar creates shared singular variable of types SharedInteger, SharedFloat, and SharedDouble for synchronization purposes. Operations over the distributed arrays and shared variables include synchronous and asynchronous read and write.

```
SharedArray.read(int index)
SharedArray.Write(int index, Object value)
```

For synchronization between threads, the SharedArray object provides lock and unlock mthods to allow user threads to realize multi-step atomic operations. It also provides a pair of read and write variants for the realization of read-modify-write operations.

```
SharedArray.read(int index, boolean lock)
SharedArray.write(int index, boolean unlock)
```

SharedArray.lock(int index)

SharedArray.unlok(int index)

The methods globalBarrier and localBarrier methods provide barrier synchronization between global threads of the entire virtual machine (across servers) and local threads (within a server), respectively. Programmers can also create their own barrier objects, via the method createBarrier, for synchronization between any group of threads. The next group of methods returns the total number of nodes of a virtual machine, local node identifier with respect to a thread, total number of local threads within a node, thread identifier within a server. The information helps programmers to distribute and redistribute data between threads.

3.2 An Example – Parallel Inner Product

In the following, ParallelInnerProduct class presents an example that performs the inner product of a vector. One each server, a number of threads, specified by programmers, will be spawned to perform the operations defined in the run method.

```
import agent.*;
import agent.dsa.*;
public class ParallelInnerProduct  extends Thread {
   private int vecSize, size;
   private SharedFloatArray vec;   // input vector
   private SharedFloatVar result;
   private DsaAgent dsa;
   public ParallelInnerProduct( DsaAgent da, int sz) {
      dsa = da;
      size = sz;
   }
   public void run() {
      vec = dsa.createSharedFloatArray("Inner Product Vector", size);
      result = dsa.createSharedFloatArray("Tmp Result Vec", numServer);
      dsa.globalBarrier();
      int blkSize = vec.length/( numNodes() * numThread() );
                                    // Assume same thr number per server
      int myMinIndex = (getNodeId*numThread+ localThreadId)*blkSize;
                                    // Assume block decomp.
      float sum = 0.0;
      for (int k = myMinIndex; k < myMinIndex + blkSize; k++)
         sum  + = vec.read(k) * vec.read(k);
float res = result.read(true);          // locks & reads result for atomic op.
      result.write(res+sum, true);  // write & unlocks the shared variable
      dsa.globalBarrier();

      if ( dsa.mainThread() )
          System.out.println(result);
    }
}
```

4 DSA Runtime Support

Recall that the DsaAgent is responsible for local and remote access to SharedArray and SharedPmtVar objects and for handling coherence protocols between replicated data items. The DSA agent is run as a daemon thread to take advantage of the lightly-loaded processors within a SMP node. Benefits from using threads to realize one-sided communication primitives on SMP clusters were widely recognized in recent research [7][9][13].

4.1 DSA Virtual Machine

A DSA virtual machine consists of a number of DsaAgents running on each machine. We designate one of the DsaAgents as a root. During the creation of a virtual machine, all newly created DsaAgents report to the root with information about the number of threads assigned by programmers. Once the virtual machine is set up, the root replies to all other DsaAgents with the configuration of the virtual machine (i.e. the number of nodes, number of threads per node, node IP addresses, etc.).

On receiving an access request to an element of a SharedArray from an application thread, the local DsaAgent determines the owner of the element according to its index and then contacts the remote DsaAgent. To tolerate remote access latency, the DsaAgent fetches a chunk of data items at a time and caches them on non-owner sites. The chunk size can be specified in the method createSharedArray. The DsaAgent maintains a cache, which is to be shared by all the local threads.

The DsaAgent deploys an SCI-like directory-based invalidation protocol to ensure cache coherence [6]. As shown in Figure 2, the agent owning a block maintains a linked list of sharers for the block and a pointer to the head of this list. The head agent has both read and write permission on its cached block whereas the others have only read permission. The DsaAgent for a local write will invalidate remote copies. A remote write will first request ownership of the block from its current owner located by tracing the ownership trail. Invalidation occurs after the current owner grants the ownership. The DsaAgent provides a method to enable or disable this mobility of block ownship. By default, mobility is enabled for shared arrays and disabled for shared primitive variables. The methods enableMobility and disableMobility change the default settings.

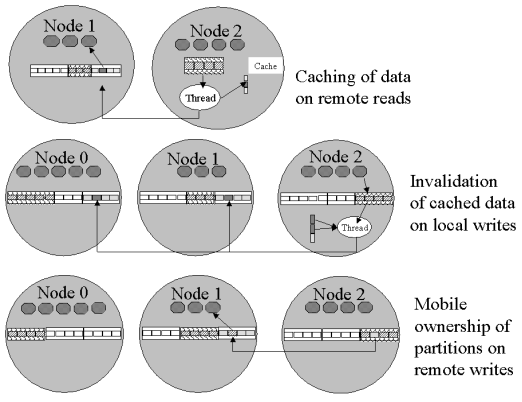

Figure 2. Mobile ownership of partitions

4.2 Mobility of the DsaAgent

Throughout the lifetime of a parallel computation, it may exhibit varying degree of parallelism and imposes varying demands on the resources. Availability of the computational resources of its servers may also change with time, in particular, in multiprogrammed settings. In both scenarios, a virtual machine must be reconfigured to adapt to the change of resource demands and supplies. Mobility of the DsaAgent simplifies the reconfiguration process.

For mobility, the DsaAgents extend Java's UnicastRemoteObject for object serialization and implement the Agent interface. During agent transfers, Java uses RMI to serialize the entire Agent object and all of its references. At the end of transfer, Java casts the serialized byte stream back into a DsaAgent object. The Agent interface provides the following method structure for mobility.

```
public interface Agent {
   public AgentId getAgentId(); // return a unique Agent id.
   public void initialize(AgentMonitor am);
                        // initializes the agent at a new Agent Host
   public void start();      //  starts the agent
   public void stop();       // stops the agent to prepare for transportation
}
```

The AgentMonitor object on each machine provides handlers to move DsaAgents between AgentMonitors. The handers enable external agents or centralized resource managers (as we will see in Section 4.3) to initiate the moving process. The AgentMonitor implements an interface containing the following handlers.

```
public boolean requestTransfer(agentId, agentMonitor)
public void beginTransfer(agentId)
public Agent transferAgent(agentId)
public void endTransfer(agentId)
```

The transfer protocol begins by invoking the requestTransfer method on the new node, which uses the parameter agentId to specify which agent to acquire and the parameter agentMonitor to specify the agent's current monitor. The new AgentMonitor transfers the agent by calling the remaining three methods on the current AgentMonitor (beginTransfer, transferAgent, and endTransfer). After acquiring the agent, the new node calls the Agent's start() method. However, since the DsaAgent is a passive object, it provides an empty implementation of the start method.

The AgentMonitor also allows for cloning of an agent and all of its shared data partitions. The cloning protocol is almost identical to transfer protocol. It contains the methods as follows. The methods beginTransfer and transferAgent work in the same way as they do in transfer protocol.

```
public boolean requestClone(AgentId, AgentMonitor)
public void beginTransfer(AgentId)
public Agent transferAgent(AgentId)
public void endCloning(AgentId)
```

4.3 External Mobile Control

The DSA system can be used standalone or be integrated with other run-time systems. Figure 3 shows a wide area parallel computing infrastructure, TRAVELER [17], which integrates the DSA run-time support system for parallel computing on a cluster of servers. TRAVELER is a mobile agent based computing infrastructure on the Internet. Unlike other Java applet-based "pull" computing infrastructure [3][4], it relies on mobile agent technologies to realize ubiquitous "push" computing. It is essentially an agent oriented broker system. The broker executes trades between clients and servers. Supported by the DSA runtime system, the broker can form a parallel virtual machine out of the available servers upon receiving an agent task. The user agent is cloned for each server and executed on the virtual machine independently of the broker. The virtual machine can be reconfigured under the request of the broker.

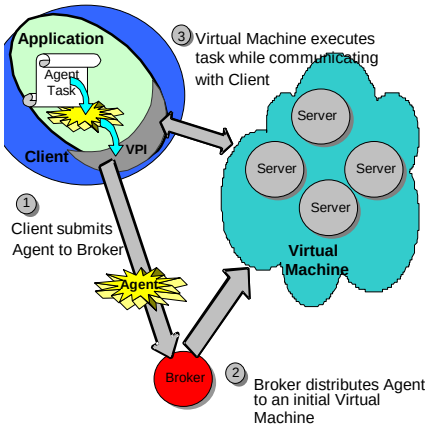

Figure 3. Architecture of Traveler

5 Experimental Results

The DSA runtime support system was implemented based on Java's remote method invocation(RMI) and object serialization facilities. The RMI facility allows RPC-like access to remote objects. Together with the object serialization facility, it supports mobile behaviors and features flexible security policies.

We conducted a number of experiments on the DSA system on a cluster of three SUN Enterprise SMP Servers. One machine is 6-way E4000 with 1.5 Gbytes of memory and the other two are 4-way E3000 with 512 Mbytes of memory. Each processor module has one 250MHz UltraSPARC II and 4 Mbytes of cache. The machines are connected through a Fast Ethernet switch. All codes were written in Java 2 and compiled with Just-in-Time compiler. They were run in a native thread mode.

5.1 Overhead of the DSA Read and Write

We first measured the access time of an array item via the DSA within an SMP and across SMP servers. We considered an array of 10,000 integers that were distributed equally among ten threads. The threads were run on a DSA virtual machine on two SMP servers (with a configuration of 6-4 processors).

Figure 4 presents remote, local access time, and average access time. The average access time was measured by scanning the whole array from the beginning to the end. The figure shows that a remote read (or write) takes 3.6 (4.3) milliseconds. It is huge compared with 1.4 (1.4) microsecond for a local read (write). Note that the remote access cost is attributed to DsaAgent and RMI implementation. For a breakdown of the cost, we present the time required for a remote read/write in RMI without involving DsaAgent in the figure (1.86ms for a read and 1.88ms for a write). From the figure, it can be seen that the DsaAgent overhead accounts for approximately half of the total remote access cost. Due to the DSA data caching and mobile ownership migration, however, the average access time is reduced to 5 to 10 microseconds.

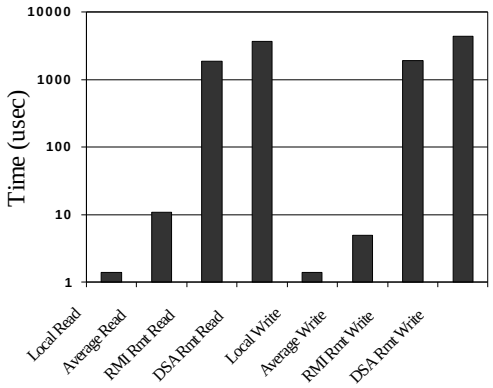

Figure 4. Cost of local and remote access over DSA

Success of the DSA mechanism also relies on the performance of synchronization operations. Instead of viewing the processors on a flat network, we implemented a hierarchical barrier synchronization across servers by designating one of the local application threads to communicate with remote threads. It was tested that local barrier and remote barrier took 3 milliseconds and 7 milliseconds, respectively. The cost of a local barrier increased with the number of threads. The overhead of remote memory access led to a big jump in cost for global synchronization

5.2 Distributed LU Factorization and Sorting

We evaluated the overall performance of the DSA in two applications: odd-even sorting and LU factorization. The odd-even sorting algorithm sorts n elements in n phases, alternating between odd and even phases, each of which requires n/2 compare-exchange operations. The LU factorization algorithm decomposes a matrix into a product of lower and upper triangle matrices. We started the DSA evaluation with these two applications because they exhibit simple algorithmic structures. their parallelization requies support

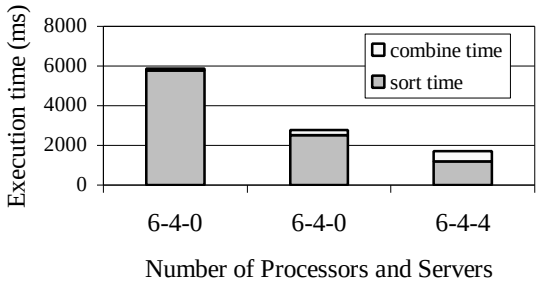

Figure 5. Parallel Sorting Time on Multiple Servers

The parallel sorting algorithm distribute the input array in a block decomposition way. Threads proceed independently over their array partitions. They are then synchronized to combine their sorted results in parallel. Figure 5 shows the total sorting time of an array of 10,000 integers on virtual machines with one, two and three SMPs. The figure clearly indicates the performance improvement due to the use of multiple servers. Figure 6 shows the breakdown of the execution time in a cluster of three SMPs. It demonstrates again the efficiency of DSA's thread synchronization.

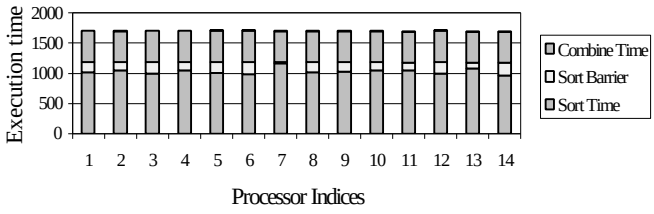

Figure 6. Breakdown of the sorting time on 3 servers

The parallel LU factorization algorithm employs a simple row-wise block decomposition. Presently, the DSA system supports only one-dimensional shared arrays. Two or higher dimension arrays must be transformed into a linear array in access. Figure 7 shows the total execution time for a double matrix of 256 x 256 on virtual machines of one, two, and three servers. Each server runs the same number of threads as the residing processors. The figure clearly shows the benefits from a parallel computing over DSA virtual machines with multiple servers.

For comparison, we include the results from implementations in C/MPI and Java socket. From the figure, it

can be seen the C/MPI version is expectedly faster than Java implementations. Comparing the results from Java socket implementations, we found that the DSA run-time system incurs no more than 30% overhead, although DSA was implemented in RMI.

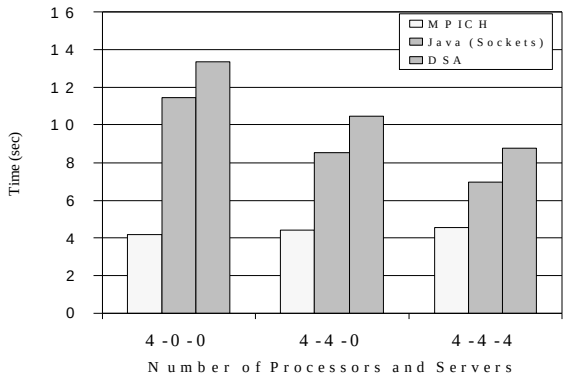

Figure 7. LU factorization of 256 x 256 double matrix

6 Conclusions

In this paper, we have presented a Distributed Shared Array (DSA) runtime support system to support Java multi-threaded programming on clusters of SMPs. The DSA programming model exposes the cluster's hierarchical organization to programmers and allows them to explicitly control data distribution while relieving them from orchestration of inter-node communications at run-time. The DSA system was developed as a mobile agent so that the DSA-based virtual machine can be reconfigured to adapt to the varying resource supplies or demands. It was implemented in Java .

We have demonstrated the programmability of the DSA system in parallel sorting and LU factorization problems on a cluster of Sun Enterprise servers. Although current prototype was presented as a proof-of-concept and has not been deliberately refined, benefits from DSA-based cluster computing were observed in both applications.

The DSA system presently supports only one-dimensional shared arrays. We are extending the system for multidimensional arrays so as to enable programmers to exploit locality in their algorithms. Considering the cost of RMI and object serialization dominates the time for remote memory access, we are implementing a RPC-like communication layer to replace RMI for remote memory access on clusters. We will also develop adaptive applications to exploit the mobility of the DSA system.

References

[1] C. Amza, et al. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, pages 18—28, February 1996.

[2] M. Baker, et al. mpiJava: A Java interface to MPI. *In Proceedings of the First UK Workshop on Java for High Performance Network Computing*, *Europar* 1998.

[3] A. Baratloo, et al. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems.* September 1996.

[4] B. Christiansen, et al. Javelin: Internet-based parallel computing using Java. *Tech. Report, UCSB*, 1997.

[5] D. Culler, et al. Parallel programming in Split-C. In *Proc. of SC'93*, pages 262—273.

[6] D. Culler, J. Singh, A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Pub. 1998.

[7] B. Falsafi and D. A. Wood. Scheduling communication on a SMP node parallel machine. *In Proc.of HPCA'97*, February 1997.

[8] A. J. Ferrari. JPVM: Network Parallel Computing in Java. I*n Proceedings of ACM 1998 Workshop on Java for High Performance Network Computing*, 1998.

[9] L. A. Giannini and A. Chien. A software architecture for global address space communication on clusters: Put/Get on Fast Messages. I*n Proc. of the 7th IEEE HPDC*, July 1998, pages 330—337.

[10] H. Karl. Bridging the gap between distributed shared memory and message passing. In *Proc. of ACM Workshop on Java for High-Performance Network Computing*. February 1998.

[11] P. Keleher. CVM: The coherent virtual machine. *University of Maryland, CVM Ver.2.0*, 1997.

[12] K. Li and P. Hudak. Memory conherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321—359, November 1989.

[13] B.-H. Lim, P. Heidelberger, P. Pattnaik, and M. Snir. Message proxies for efficient, protected communication on SMP clusters. Iin *Proc. of HPDC'1997*.

[14] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A portable shared-memory programming model for distributed memory computers. In *Supercomputing*'94, 1994.

[15] S. Roy and V. Chaudhary. Strings: A high performance distributed shared memory for symmetric multiprocessor clusters. In *Proc. of the 7th IEEE HPDC*, August 1998.

[16] E. Speight and J. Bennett. Brazos: A third generation DSM system. In *Proceedings of the First USENIX Windows NT Workshop*, August 1997.

[17] B. Wims and C. Xu. Traveler: A mobile agent infrastructure for global parallel computing. In *Proc. of 1st Joint Symposium: Int. Symp. on Agent Systems and Applications (ASA'99) and Third Int. Symp. on Mobile Agents (MA'99*), October 1999.

[18] A. Yu and W. Cox. Java/DSM: A platform for heterogeneous computing. *In Proceedings of ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.