Distributed Shared Array: An Integration of Message Passing and Multithreading on SMP Clusters

Ramzi Basharahil, Brian Wims, Cheng-Zhong Xu[1], and Song Fu
Department of Electrical and Computer Engineering
Wayne State University, Michigan 48202

Abstract. This paper presents a Distributed Shared Array runtime system to support Java-compliant multithreaded programming on clusters of symmetric multiprocessors (SMPs). As a hybrid of message passing and shared address space programming models, the DSA programming model allows programmers to explicitly control data distribution as so to take advantage of the deep memory hierarchy, while relieving them from error-prone orchestration of communication and synchronization at run-time. The DSA system is developed as an integral component of mobility support middleware for grid computing so that DSA-based virtual machines can be reconfigured to adapt to the varying resource supplies or demand over the course of a computation. The DSA runtime system also features a directory-based cache coherence protocol in support of replication of user-defined sharing granularity and a communication proxy mechanism for reducing network contention. We demonstrate the programmability of the model in a number of parallel applications and evaluate its performance on a cluster of SMP servers, in particular, the impact of the coherence granularity.

Keywords: Cluster of symmetric multiprocessors, distributed Shared Array (DSA), distributed shared memory (DSM), and Parallel Programming Model.

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 overlay the nodal operating systems. It is known that parallel programming involves four main steps [9]: 1) Decomposing the problem into a smaller unit of tasks that can be executed concurrently; 2) Assigning the tasks onto processing entities (process or a thread); 3) Orchestrating the execution of the entities, focusing on their communication and synchronization as needed; 4) Mapping the processing entities to physical computing resources (processors) to carry out the actual computations. Together, the decomposition and assignment steps are often referred to as partitioning. Partitioning is often algorithmic in nature and is independent of the underlying architecture and programming models. The mapping process tends to be platform-dependent. The orchestration is determined by parallel programming models. It determines how to organize the data structure, whether to communicate

[1] Correspondence author: Cheng-Zhong Xu, Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202. Email: czxu@wayne.edu

implicitly or explicitly among processes/threads, and how much of an effort the programmers should spend to orchestrate and synchronize the inter-process communications.

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 demonstrated hard for parallel programming because programmers are required to explicitly distribute data across processes' disjoint address spaces and schedule inter-process communications at run-time. Shared address space programming model like OpenMP[6] assumes shared variables for communication and synchronization between threads on SMP and NUMA parallel computers. There are software-based distributed shared memory (DSM) systems, such as page-based Ivy [19] and TreadMark [1] and object-based Orca [3], in support of shared address space programming on clusters; see [24] for a review of early work on this topic. DSM systems simplified parallel programming with a compromise of efficiency. Since a CLUMP employs a deep 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.

Recent research focus is on the development of multithreaded DSM systems to explore the potential of SMP nodes. Examples include Brazos[27], CVM [18], SilkRoad [23] and String [26][16]. 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 bias the first and their actual performance is yet to be seen. Previous studies on high performance cluster computing have showed that performance of a parallel program is critically dependent on data distribution and locality would be. The deeper the memory hierarchy is, the greater the impact of data distribution and locality. It is also known the complexity of message passing programming is mainly due to the orchestration of processes at run-time [8][34]. This paper presents a Java-based distributed shared array (DSA) runtime support system for a tradeoff between the two objectives in another way. It relieves the programmers from error-prone runtime scheduling of inter-node communications while giving them more control over data distribution.

There are many Java-based parallel programming environments for high performance cluster computing. Examples include MpiJava[2], JPVM[11], Java/DSM[37], Jackel[31], JESSICA[38], and JavaParty[25]. In comparison with these parallel programming environments, the Java-based DSA system has the following unique features.

- It is tailored to CLUMPs with a disclosure of nodal architecture to the programmers for a better tradeoff between programmability and efficiency. Programmers are responsible for thread creation and assignment. It supports replication of user-defined array partitions to reduce remote access latency;
- It provides a low level communication and high level scheduling proxy mechanisms for reducing network contention and accelerating cache coherence protocols;
- 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 so as to adapt to the change of resource supplies or requests.

The rest of the paper is organized as follows. Section 2 presents the DSA architecture and programming model. Section 3 shows the implementation details of the DSA runtime support system.

Section 4 presents experimental results on a cluster of Sun Enterprise servers. Section 5 presents related work and Section 5 summarizes the results with remarks on future work.

2 The DSA Architecture and Programming Model

2.1 DSA System Architecture

The DSA system provides a Java-compliant programming interface to extend multi-threaded programming on clusters of SMPs. Since the array data structure is the most common data structure used by scientific and engineering application programs, it becomes a primary building block of the DSA system. The array is partitioned into smaller data blocks and distributed across cluster nodes. The DSA run-time system provides a single system image across the cluster for the distributed shared array. It relieves programmers from run-time scheduling of inter-node communications and the orchestration of the data distribution. Remote data access is supported by the DSA run time system and hidden to application programs.

As shown in Figure 1, the DSA runtime system comprise of a main API component known as the *DsaAgent*. A distributed virtual machine consists of a group of *DsaAgents*, each running at a cluster node. Multiple virtual DSA systems could co-exist spanning and sharing multiple SMP nodes, but each system has its own distinct group of *DsaAgents*. The *DsaAgents* are responsible for local and remote access to shared objects. A distinguished *DsaAgent* acts as *DsaMonitor* that holds the responsibility of managing the interconnected nodes.

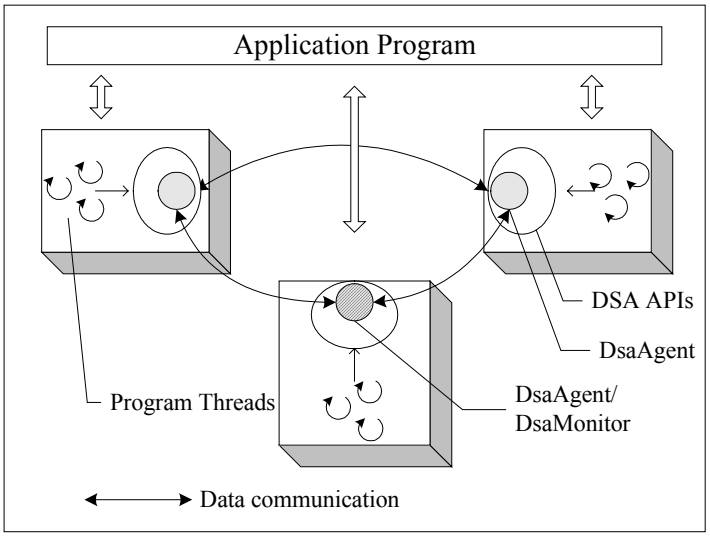

Figure 1 The DSA Architecture

We summarize the characteristics of the DSA programming model as follows:

- The DSA model supports multithreaded programming in a single-program-multiple-data (SPMD) paradigm. The concurrent threads running on different cluster nodes share arrays distributed among the nodes.

- Threads in the same node can communicate through local shared variables, while communicating with other threads on different nodes through DSA synchronous remote operations. On other hand, accessing shared array follows the shared address space programming model. The threads can access any parts of the shared array using loads (read) and stores (write) operations with the same array indices as in the sequential program.
- Access to any array elements is transparent to parallel programs. It is the DSA run-time system that supports remote array access. The DSA system also supports replication of user-defined array partitions to reduce remote data access cost.
- The application program specifies how a matrix is physically distributed among the nodes and how the array layout is assigned to running threads. It also controls the granularity of cache coherence.

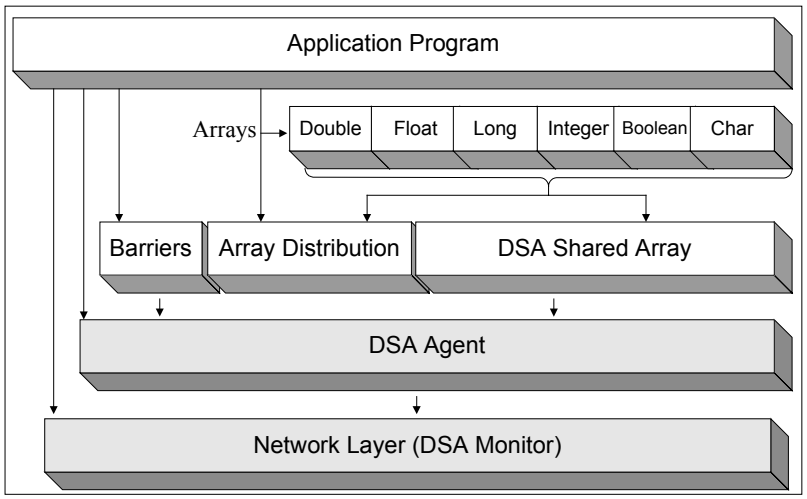

Figure 2: DSA API components

2.2 DSA APIs

The DSA programming model relies on two first class objects: DsaAgent and SharedArray. It implements a set of primitive operations are for access to use-defined shared arrays and management of the shared data structure. Figure 2 shows the DSA API organization.

A DSA virtual machine is consisted of multiple *DsaAgents*, running on each cluster node. DsaAgent implements an Agent interface. An agent is a special object type that has autonomy. It behaves like a human agent, working for clients in pursuit of its own agenda. Details of the Agent interface will be discussed in Section 3.4. During the startup of a virtual, the application program has to initialize an instance of the *DsaAgent* by supplying the necessary configuration information about the virtual machine. The configuration information includes the nodal IP addresses, the number of threads running on each node (under the control of a DsaAgent). Due to the hierarchical organization of SMP clusters, we refer to properties related to individual nodes as "local". We also designate node-0 as a monitor node for the management of the virtual machine. The *DsaAgent* residing on the monitor node is designated as a *DsaMonitor*. All newly created *DsaAgents* need to report to the *DsaMonitor* with information about their current configurations. Once the virtual machine is set up, the object *DsaNetwork* provides methods for

access to the virtual machine configuration information. For example, methods *getLocalThreadCount*, *getGlobalThreadCount* return the number of threads locally and globally and method *getDsaNetIPs* returns the node IP address.

The DSA system defines two distributed variables: SharedArray and SharedPmtVar. The SharedArray objects are created by method *createSharedArray*. Since Java doesn't support operator overloading, the *DsaAgent* actually provides extensions of *SharedArray* for different element types. For examples, methods *createIntSharedArray* and *createFloatSharedArray* return references to distributed shared integer and float arrays respectively. SharedArray objects can be distributed between threads in different ways. Currently, one and two dimensional block dimensional block decompositions are supported. In addition to the way of block decomposition, programmers can also specify array partition size as the granularity of coherence for data replication during the creation of a shared array. 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 distributed shared arrays and shared primitive variables include synchronous and asynchronous read and write. For efficiency, the DSA system also provides APIs, including *ReadRemotePartition* and *WriteRemotePartition*, for access to a block of array elements of different sizes. These APIs are invoked within the spawned threads of a parallel program. They are trapped by the *SharedArray* to determine whether a remote or local access is required or if there is a valid cache of the array partition. The *SharedArray* then returns the required array elements.

In addition to read/write operations, the DSA system provides APIs for barrier synchronization between application threads. *LocalBarrier* and *GlobalBarrier* are two built-in objects for this purpose. The local barrier is used to synchronize threads residing on the same node and the global barrier is for threads in the entire virtual system. Programmers can also create their own barrier objects, via the method *createBarrier*, for synchronization between any group of threads. The application program has direct access to the distribution component of the DSA system. That allows more optional control over the distribution/caching mechanism performed by the DSA. APIs such as *getPartition*, and *getCacheProtocol* provides information about the cache distribution among the DSA nodes. *requestPartitionCopy*, *acquirePartition* to perform cache acquisition and ownership transfer.

In the following, we illustrate the DSA programming model through an example of parallel inner product.

2.3 An Example

Execution of a DSA program involves three main steps: (1) to create a DsaMonitor that manages the shared array; (2) to create the required shared arrays; (3) to access the shared through read/write operations. While the first step is within the main thread of control, the second and third steps are to be invoked in the working threads of a parallel program spawned by the main thread of control.

As shown in the program skeleton in Figure 3, the main program flow will first determine the user specific parameters from the command line. These configuration parameters are used to instantiate *DsaAgents*. The DsaAgents register their services to a DsaMonitor and together form a fully connected DsaNetwork. DSA access occurs in working threads of the parallel application, which create shared arrays. A global barrier ensures the synchronization of this step. The working threads proceed and generate different access patterns of the shared array. The DsaAgents trap the array accesses and

determine, with the help of a cache coherence mechanism, if remote access is needed or the data partition is available locally. In case of a remote access, the DsaAgent contacts the remote counterparts and requests the data partition.

```
import java.util.*;
import java.io.*;
import DSA.Agent.*;
import DSA.Arrays.*;
import DSA.Sync.*;
import DSA.Distribution.*;
import DSA.Communication.*;

 public class InnerProduct {
 // call: java ArrayAccessClient (array dimension) (partitionSize) (number Of local threads)
 private static int numOfThreads = 0, arrayDimension = 0, partitionSize = 0;
 private static DsaAgent dsa = null;
 private static FloatSharedArray matrixA = null;
 private static FloatSharedArray matrixB = null;
 private static FloatSharedArray resultMatrix = null;
 protected static DsaBarrier stageBarrier = null;

 public static void main(String args[]) {
   try {
    arrayDimension = Integer.parseInt(args[0]);
    partitionSize = Integer.parseInt(args[1]);
    numOfThreads = Integer.parseInt(args[2]);
                                                  // Creat the DSA and create the shared arrays.
    dsa = new DsaAgent(numOfThreads); dsa.initialize(); // 2D arrays are created.
    matrixA = dsa.createSharedFloatArray("matrixA", arrayDimension, arrayDimension, partitionSize);
    matrixB = dsa.createSharedFloatArray("matrixB", arrayDimension, arrayDimension, partitionSize);
    resultMatrix = dsa.createSharedFloatArray("resultMatrix", arrayDimension, arrayDimension, partitionSize);
    stageBarrier = dsa.CreateGlobalBarrier("globalBarrier");
    InitializeMartixAandMatrixB();                // Will initialize the data set for both matrices.
                                                  // But not shown in the example to save the space.
    Thread threads[] = new Thread[numOfThreads];
    for (int i=0;i<numOfThreads;i++) {
     threads[i] = new Thread(new Worker());
     threads[i].start(); }
     for(int i=0; i < numOfThreads; i++)
       threads[i].join();
     dsa.finalize();
  } catch (Exception e) {e.printStackTrace();}
 }

 private static class Worker implements Runnable
 {
  public void run() {
   float sum = 0;
   arrayDistribution matrixALayout = matrixA.getArrayLayout();
   arrayDistribution matrixBLayout = matrixB.getArrayLayout();
   stageBarrier.barrier();                        // All threads across the nodes will start at the same time.
   try {
     for(int i = matrixALayout.startingIndexAtThread(); i < matrixALayout.maxIndexAtThread(); i= i++)
       for(int j = matrixBLayout.startingIndexAtThread(); j < matrixBLayout.maxIndexAtThread(); j=j++ ) {
         sum = 0;
         for(int k = 0 ; k < arrayDimension ; k++)
           sum += matrixA.read(i, k) * matrixB.read(k,j) ;
         resultMatrix.write(i,j,sum);
       }
    }
    catch(Exception e){e.printStackTrace();}
  }
}
}// end of class
```

Figure 3. An Example of DSA Parallel Program.

3 DSA Runtime Support

Recall that the DsaAgent is responsible for local and remote access to distributed arrays and for handling coherence protocols between replicated array partitions. 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 [10][13][20]. The DsaAgent features a directory-based cache coherence in support of replication of user-defined array partitions and a communication and scheduling proxy mechanism for reducing network contention. The DsaAgent is developed as a mobile agent so that the DSA based virtual machine can be reconfigured to adapt to the change of resource supplies and demands.

3.1 The DSA Cache Coherence Protocol

The CLUMP architecture is characteristic of deep memory hierarchy and non-uniform memory access cost at different levels. In particular, remote memory access across the SMP nodes often costs up to three orders of magnitude more time than local memory access within a node. An important technique to reduce the remote access time is data replication. In literature, there are many studies on data replication and its related cache coherence protocols. Most of the coherence technologies assumed a fixed coherence unit. By contrast, the DSA system support replication of user-defined array partitions and the partition size can be specified during the creation of a shared array according to the virtual machine configuration. For the purpose of load balancing, the partition size is often set to the array size divided by the total number of participating threads.

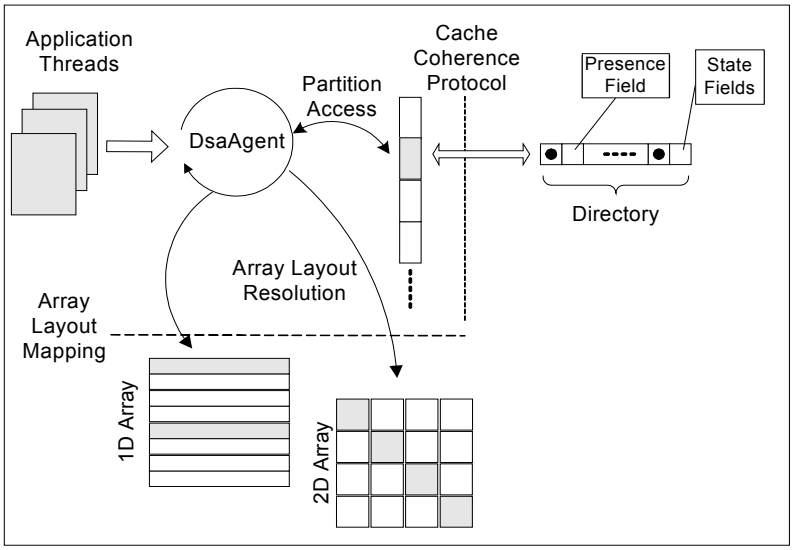

Figure4: DSA Directory Based Cache Coherence

The DSA system follows a directory based cache coherence protocol to manage cluster wide data replications. In this protocol, each partition is designated a home node. A home node tracks the sharers of its partition in a directory structure. Due to the tightly coupled environment of interconnected cluster nodes, the home node will update that state of the partition according to the remote operations that it

receive. The state of a cached partition is determined explicitly in the cache directory, as shown in Figure 4. A unique ID, known as the partition index, is assigned to each *SharedArray* partition. The cache directory keeps track of each partition status using 2 bits on each partition entry. Partition status can be Shared, Exclusive, and Invalid. Shared status means the current node has up to date copy of the partition but other nodes share it. Exclusive status means the current node is the only node that has an up to date copy of the partition. Invalid status means some other node is changing the partition and the current copy is out of date. Each home node keeps track of who has an exclusive copy and who are the sharers for its assigned partitions by using a set of presence bits. Each bit corresponds to a unique DSA node..

On receiving a request for a *SharedArray* element, the local *DsaAgent* will determine the availability of the requested element. It will check through the cache directory wither a valid copy of the requested partition exists in the local memory. The *DsaAgent* delegates the call to the cache coherence algorithm to determine whether the read or write access is a cache hit or miss. Upon a cache miss, a partition request call propagates to the home node. The home node depending on the status of the partition in request, makes share transaction on request for exclusive copy, update transaction on shared reads, or invalidation transaction on writes. Owner ship of partitions movies to exclusive node on write request. The home node directory maintains the sharers nodes, and exclusive node of a partition. Figure 5 shows basic operations on the coherence protocol.

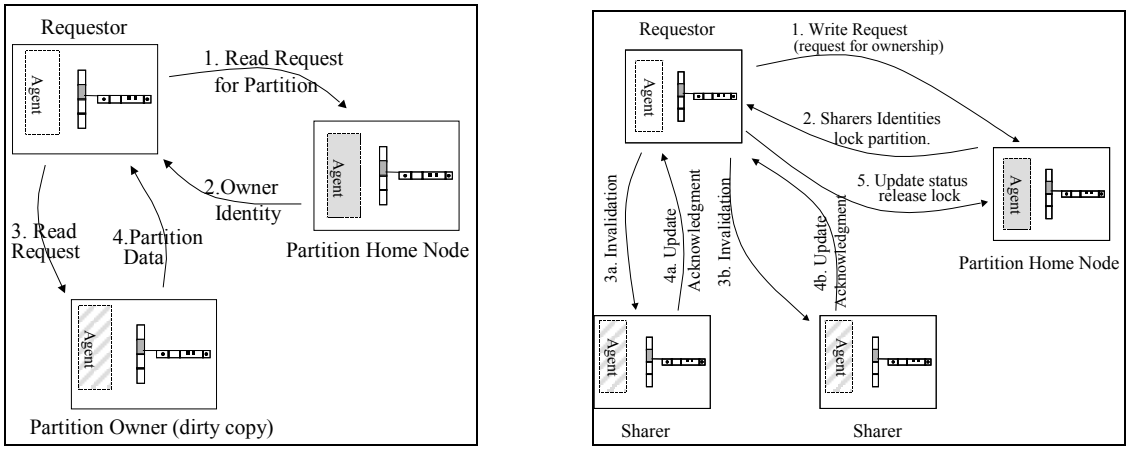

Figure 5: Cache Coherence Operations

The cache coherence protocol introduced over the cluster of SMPs generates network communication transactions. The transactions can be requests, acknowledgments, or cache partitions along with some control attributes. Although these transactions are part of the communication activities that the DSA system tends to hide and their costs are main concerns for the DSA efficiency. The coherence protocol has no control over the cost of individual network transactions. But the protocol design and algorithm has more control over the number of network transactions required to handle read or write cache misses by reducing the number of network transactions generated per cache miss operation, the demand for the network bandwidth decreases, and the contentions on the network resources.

On a read miss, instead of following the strict request-response approach introduced earlier, more efficient intervention forwarding is implemented. In the first scenario the home node reply the requester with the identity of the owner, the requester then contacts the owner to share the data, the owner also send a revision of the data to the home node. With the intervention-forwarding scheme, the home node delegates (forwards) the request directly to the owner node instead replying with its identity. As the owner reply with the shared data, the home node updates its cache and forwards the reply to the requester. With this approach the number of transactions has be reduced to four instead of five.

On a write miss, we can distinguish two different situations with different communication costs. In the first scenario, the partition cache is in the invalid exclusive state owned by a third node. In that case, the home node forwards the request of ownership to the current owner of the cache; the owner invalidates its cache state and reply with the most recent data to the home node. The home node then returns that cache to the requesting node. This case behaves just like intervention forwarding mentioned before, and although the cache state change is different; the number of transactions is the same. In the second scenario also shown in Figure 8, the cache is in the shared state, and shared by more than one node, in this case, the home node takes the necessary steps to invalidate all the sharers of that cache, upon completion, the cache owner ship handed out to the requesting node.

Besides reducing the number of network transaction per read/write misses, there are two main advantages by making the home node the center of these coherence protocol transactions.

When a request arrives at the home node, the directory coherence lookup determines if the home node has a valid cache of the requested partition in memory, in most read miss cases hopefully that will be the true, since a cache partition on shared state always guarantee a valid data on the home node, thus reducing the number of transaction even further. So taking the case of two consecutive read miss requests on the same cache partition from two nodes, the first request will generated four transactions, while the second will only introduce two, since the home node will reply with a valid cache directly and does not need to forward the request to any other node since the cache already in valid shared state.

The second advantage is ensuring consistency by the locking mechanisms associated with a cache invalidation/update on the home node, thus behaving like an atomic transaction that might consists of multiple transactions. So, if another requester from another node inquires the same cache partition, the lock on the home node insures the correctness of the data and does not allow the raises of race conditions.

Finally, we note that one of the objectives of the DSA system is to give programmers more control of data distribution while hiding the details of orchestrations for thread safe data accesses. However, a strict consistency for a global total order between read/write operations would incur extremely high more read/write overhead. Relaxed consistency is to make a good tradeoff between efficiency and programmability. In a relaxed consistency, it becomes the programmer's responsibility to make sure that the application threads have safe access into any shared objects, as they will normally do in a multithreaded programming model. This is a simple and intuitive rule because any thread safe application will follow to make sure no race condition exists for correct data access. The DSA run-time support system assumes a relaxed consistency in the access of shared array partitions. The cache coherence protocol ensures thread safe access to shared array partitions by exploiting concurrency between reads and writes.

3.2 Communication Proxy to Reduce Network Contention

In the DSA system, each DsaAgent daemon is essentially a communication proxy, which implements remote data read/write for local application threads. Since the daemon dedicates one thread communication handling, we refer to this as a fixed-scheduling model. It handles requests by receiving a request on a work queue. Upon the completion of the request, the dedicated communication proxy sends an acknowledgment to the waiting threads.

An alternative to dedicated communication proxy is float proxy, in which any application thread is capable of communication to remote nodes. As shown in Figure 6, when multiple threads have requests for remote communication (due to a cache miss or a synchronization event), one of the threads that request the partition can be designated as a proxy of its siblings. As soon as it finishes the communication, it signals the other application threads that are trying to access the same partition or waiting for the same synchronization event. Based on the locking state a partition, the coherence protocol can determine if a proxy thread has already served a remote call on the partition.

We illustrate the float-scheduling model in an implementation of barrier synchronization. A barrier operation across a group of threads is to ensure no threads can go beyond the barrier until all the threads arrive at the barrier. In small-scale SMP machines, the barrier is often implemented via a centralized approach, in which a shared variable counter keeps track of the number of arrivals. This centralized approach in the SMP clusters can lead to a lot of traffic and contention. An alternative is software tree structured barrier. The tree barrier defines two level arrival/release trees to synchronize the threads and avoid hot spots on communication resources. When a thread arrives at the barrier, it checks if it is the last one of the local group to arrive. If so, it designates itself as the local barrier proxy and communicates to central authority of the barrier via a blocking barrier call. The central authority can be the DsaMonitor of the virtual machine. Since the network communication involved in the second stage is per SMP node instead of threads, this tree-structured barrier reduces network contention and ensures good scalability.

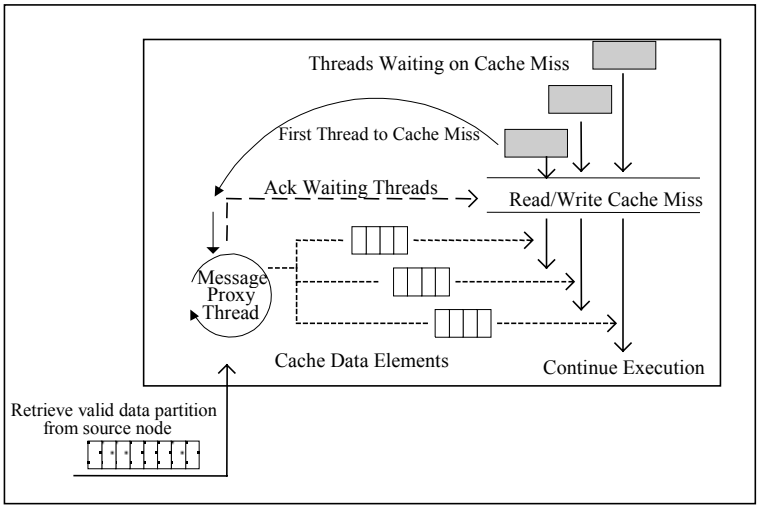

Figure 6. Communication Proxy Architecture

3.3 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 implement a mobile 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 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)
```

The DSA system can be used standalone or be integrated with other run-time systems. Figure 6 shows a wide area parallel-computing infrastructure, TRAVELER [32][35], 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, 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.

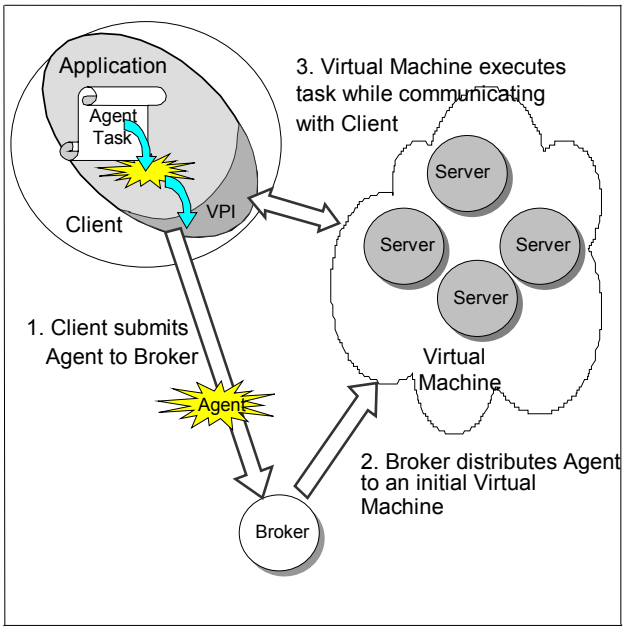

Figure 7. Architecture of Traveler

4 Experimental Results

The DSA runtime support system was implemented based on Java's remote method invocation (RMI). The RMI facility allows RPC-like access to remote objects. We conducted a number of performance evaluation experiments on a cluster of four SUN Enterprise SMP Servers. One machine is 6-way E4000 with 1.5 Gigabytes of memory and the other two are 4-way E3000 with 512 Mbytes of memory. Each processor module has one 330MHz UltraSPARC II and 4 Mbytes of cache. The machines are connected through a 100Mbps Fast Ethernet switch. All codes were written in Java, compiled in JDK 1.2 with Just-in-Time option, and run in a native thread mode.

4.1 DSA Startup Overhead

In [32][35], we focused on the mobility of the DSA in the Traveler infrastructure and tested the cost for its remote startup overhead. In this experiment, we measure the cost for the DSA construction as a standalone run-time support system. As the DSA system starts, the first thing is to create DsaAgents on participating nodes and form a DSA network (virtual machine) across the machines. When a shared array is created, the *DsaAgent* provides detailed information about the array (e.g. array size and partition size) to its cache coherence protocol for efficient management of remote data access. Figure 8 shows the DSA

startup overhead on various virtual machine sizes and the cost for creation of shared double arrays. Assume the arrays are partitioned in two-dimensional block decomposition and the partition size is 32 x 32.

	One Node	Two Nodes	Three Nodes	Four Nodes
DSA Network Startup	93 ms	430 ms	583 ms	624 ms
Creation (512x512)	104 ms	92 ms	78 ms	67 ms
Creation (1024x1024)	148 ms	127 ms	108 ms	89 ms

Figure 8: DSA startup overhead and cost for creation of shared arrays.

The figure shows that as the DSA virtual machine size increases, its construction time increases. This startup overhead includes the transmission time of the DSA network configuration information from DsaMonitor to DsaAgents and the registration time from the DsaAgents to the DsaMonitor. That is why construction of a virtual machine of 2 nodes costs much more than a single node machine. When the virtual machine size increases from 2 to 4 nodes, the startup overhead increases at a sub-linear rate because of overlaps of registrations from different DsaAgents. From the figure, it can also be seen that the cost for creation of shared arrays decreases as the virtual machine expands. It is because all the partitions are equally distributed among the cluster nodes and the number of array partitions per node decreases as the virtual machine size increases.

4.2 DSA Read/Write Cost

The DSA run-time support system provides a high-level programming abstract, built over the RMI communication layer. It is essential to know read/write overhead due to the DSA support, in comparison with RMI communication cost. Moreover, we developed an RPC wrapper over Java sockets to perform the same DSA read/write operations.

In the case of RMI communication, we tested the cost for remote read/write for a number of times, each with access to different block sizes, and took an average of the costs. In order to perform equivalent remote access in the DSA system, all read and write operations over a shared array have to be cache miss. To the end, we accessed the first element of each partition and set the stride of consecutive data accesses to be the shared array partition size. At the beginning of the test code, one DsaAgent walked through all the partition and requested exclusive ownerships for the partitions. Then, the second DsaAgent started the actual test and all its references were guaranteed to be read/write miss. The RPC wrapper was implemented through a two-way Java socket communication, which performs solely read/write operations of various block sizes.

Figure 9 shows remote read/write cost via the DSA system, in comparison with the approaches of RMI and RPC wrapper. From the figure, it can be seen that the DSA read/write miss incurs small percentages of overhead over the RMI access of partitions of medium and large sizes. For large partitions of 128x128 size, the DSA read/write miss incurs a marginal overhead. But for small partitions of 8x8 and 16x16 sizes, the DSA remote access incurs overhead as high as 60 percentages.

From the figure, it can also be seen that the RPC wrapper over two-way Java socket communication is more expensive than RMI. The RPC wrapper is based on two object transfers between client and server over Java socket and each requires marshaling and un-marshaling of serialized objects over the network. The cost for marshaling/un-marshaling is the dominant factor for both RMI and Java socket as they both

rely on object serialization. But in the RMI case, the remote read/write needs to send array indices of primitive types. Since the transferring of primitive data needs no serialization, its cost is much less than the object marshaling/un-marshaling as required by the RPC wrapper.

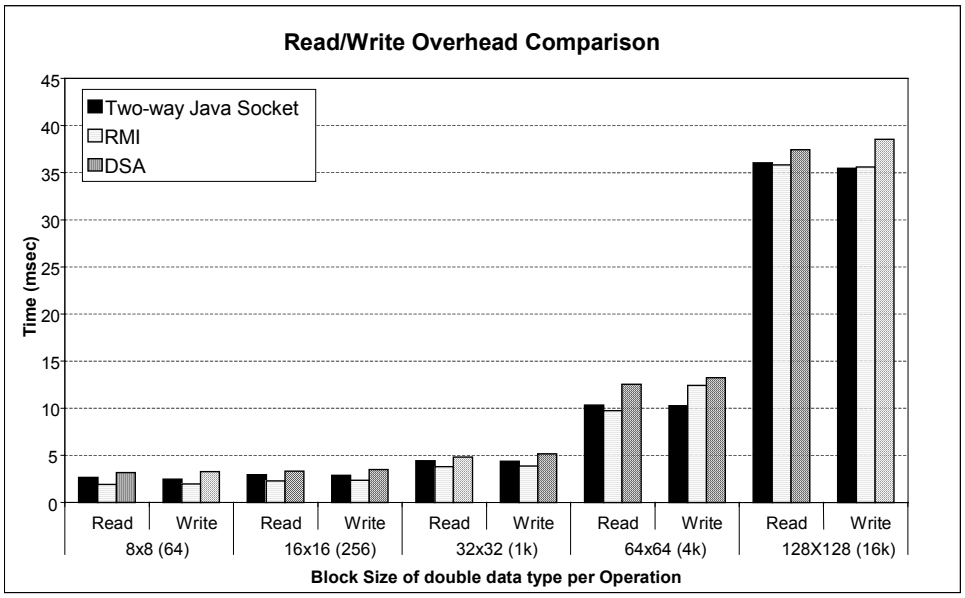

Figure 9. Remote Read/Write Overhead in DSA, RMI, and Socket Wrapper

4.3 DSA Average Write

In the third experiment, we tested the average cost for data access due to the impact of partition replication. Since the write operation involves cache invalidation protocol, we considered writing of single array elements of a shared double array. We assumed a requestor node to be performing all the writes and waiting until all other participating nodes read shared array elements and become active sharers. The requestor node then wrote elements into the shared array consecutively by a stride of various sizes so as to ensure that not all writes would be hit once an array partition becomes exclusive. The requestor kept performing writes for one second. The test was repeated under different configurations with different partition size, strides and the number of participating nodes:

- The number of participating nodes started from two up to four in consistent with the rest of the experiments;
- The partition size started from 8x8 and increases up to 32x32;
- The stride of consecutive write indices started from one up to partition size in a dimension. For example, for the partition of size 8x8, the maximum stride is set to 8.

Figure 6 shows the average cost for a write in different configurations. It reveals that the average cost for a write increases as the number of participating nodes. For example, in the case of stride of 2 and block size of 1024, as the number of participating nodes increases from 2, 3 to 4 nodes, the average write cost increases from 4.7ms, to 6.5ms and 9.4ms, respectively. This is because the partition was set to be shared by all the nodes, except the requestor, before its write. The more the sharers are, the more nodes are involved in cache-coherence protocol for invalidation.

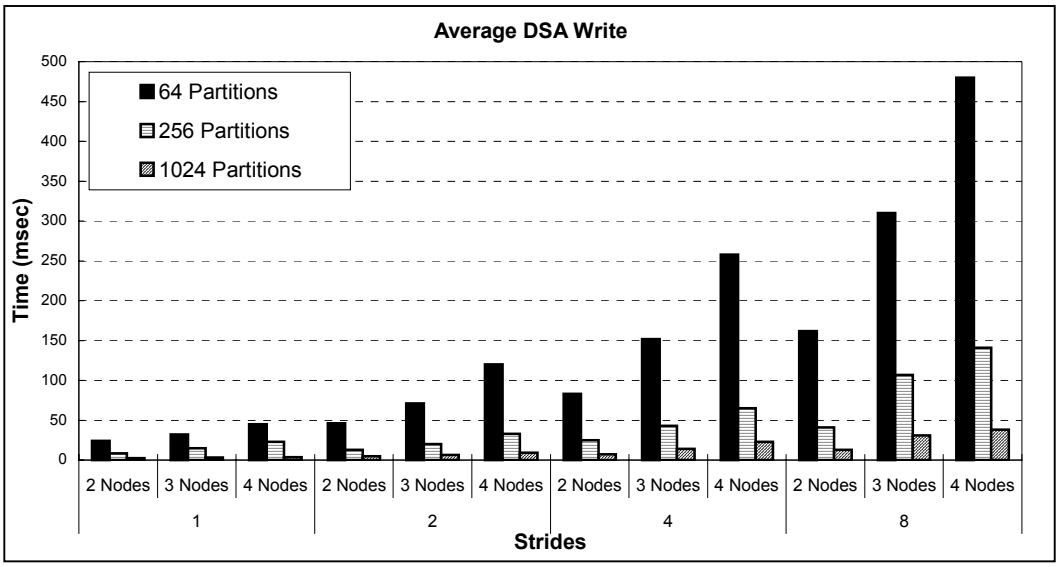

Figure 10. Average cost for a write to distributed shared arrays.

The figure also shows the advantage of large partition. As the partition size increases, there would be more cache hits than misses. Consequently, there would be fewer demands for cache invalidation and less average write cost. For example, in a 4-node virtual machine, the average time for writes in a stride of 2 is reduced from 120 to 9.4 microseconds as the partition size increases from 64 to 1024.

4.4 Tree Structured Barrier Performance

In addition to the cost for data access, performance of the DSA system is also critically dependent upon the cost for synchronization. Instead of viewing the processors on a flat network, we implemented 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 for a local barrier increases with the number of threads and the remote data access overhead leads to a jump in cost for global synchronization. The test was conducted by making a large number of consecutive global barrier calls by each participating thread in an application.

The results in Figure 11 demonstrate the poor scalability of the centralized barrier with respect to both the virtual machine size and the number of threads. For example, with four nodes and four threads running at each node, the centralized barrier costs three times more than the tree-structured barrier in a float-scheduling mode. With one thread per node, both the centralized and the tree-structured barriers are running equivalently. As we scale with more threads and more nodes, the tree-structured based technique demonstrates a better scalability because it avoids hot spot in communication.

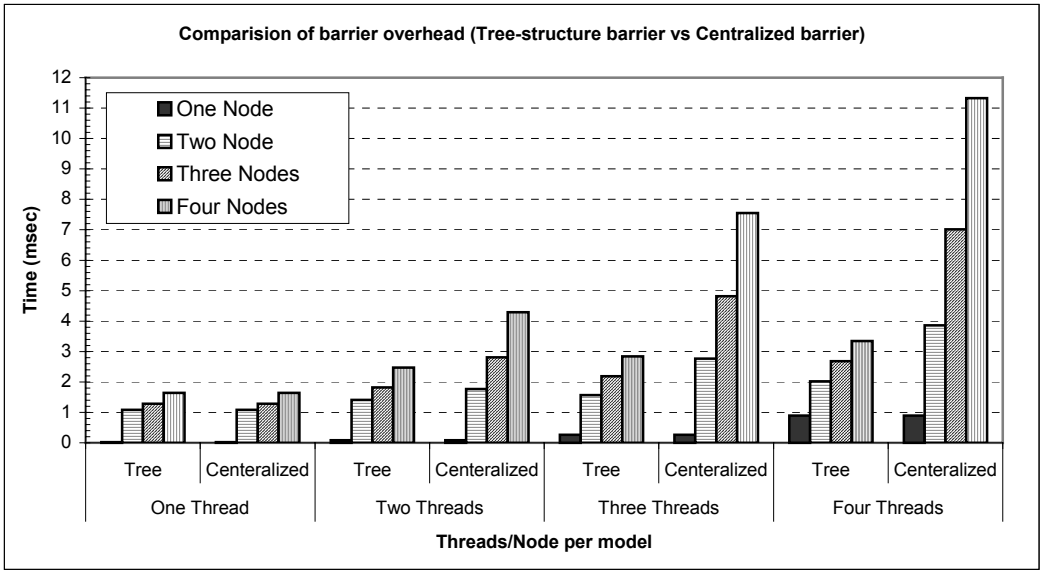

Figure 11. Barrier Synchronization Performance

4.5 Distributed LU Factorization and FFT

Micro-benchmarking aside, we evaluated the overall performance of the DSA system in two applications: LU factorization of matrices and Fast Fourier Transformation (FFT).

4.5.1 LU Factorization

LU factorization is a kernel for the solution of systems of linear equations. It loops over the matrix diagonal elements and performs pivotal row scaling and submatrix calculations at each iteration. We considered a double array of size 1024x1024. We exploited parallelism within the iteration of the out-most diagonal loop and experimented with a two-dimensional block cyclic decomposition approach with various block sizes for balanced task assignment.

Figure 12 presents the results in different configurations of the DSA virtual machine. It is known that the partition (or block) size determines the ratio of computation to communication as well as the overhead of cache coherence protocol. Decomposition with small size partitions implies less calculation before a thread proceeds into a communication phase, possibly requesting data from threads in remote nodes. This explains the poor scalability of the algorithm for small partitions of 8x8 and 16x16. As partition size gets larger, the ratio of computation to communication increases and the average remote access time decreases, as shown in Figure 4.3. However, large partitions lead to a severe load imbalance problem. For examples, even in a single node machine (4-way SMP node), partition size 64x64 yields worse performance than partition size 32x32, both assuming 4 threads running concurrently. With the increase of the virtual machine size and the number of participating threads, the load-balancing problem becomes even more severe. Meanwhile, each thread gets fewer tasks to do. Figure 12 shows the partition size of 32x32 gives a relatively good trade off, although the algorithm gains marginal improvements due to cluster computing. The algorithm showed a performance gain of 27% when the virtual machine expanded from a 4-way SMP node to two SMP nodes.

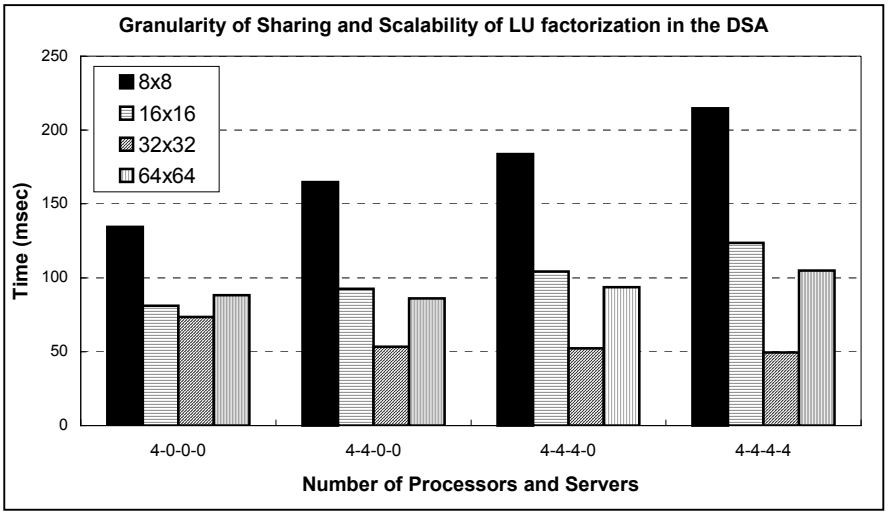

Figure 12. LU Factorization of Matrices of Various Sizes

There are many reasons for the poor overall performance. To isolate the factors of the DSA overhead, cluster architecture, and Java programming environment, we compared the LU factorization performance of the DSA version with a Java socket implementation and a MPI/C version and presented the results in Figure 13. The DSA overhead over the pure Java socket implementation is due to: 1) RMI communication, and 2) cache coherence protocol. As the virtual machine expands, there are more DSA nodes sharing cached partitions, which leads to a higher access miss rate. This figure also demonstrates the performance gap between Java and MPI/C languages.

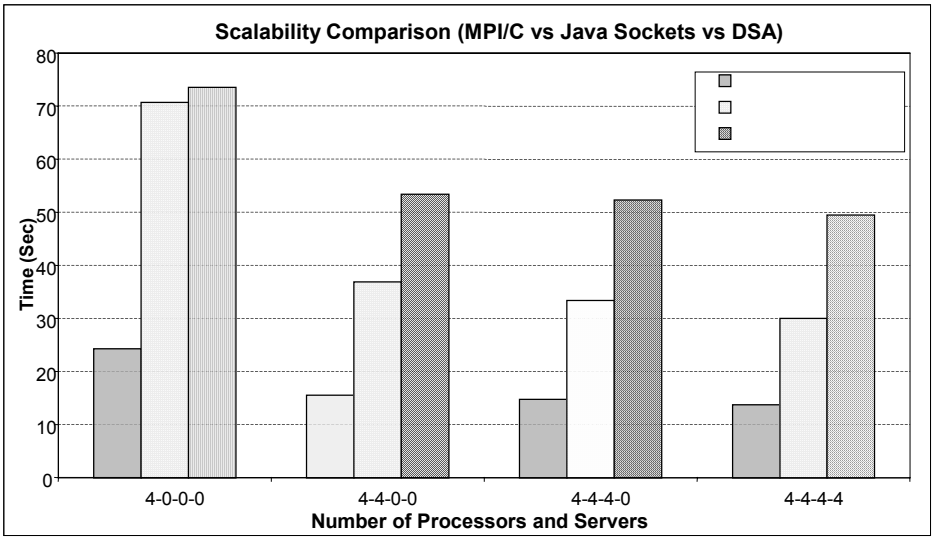

Figure 13. LU factorization of 1024x1024 double matrix

4.5.2 FFT Application

FFT is one of the fundamental problems in digital signal processing. It is often used in many scientific applications such as time series and wave analysis, convolution and image filtering. FFT algorithms compute the discrete Fourier transform on N = (2n) points in O(nN) time. An n-point FFT algorithm involves n stages of computation. In each stage, every point would be updated once, involving one complex multiplication and one complex addition. In a multithreaded FFT, each thread is assigned a portion of the data set and performs independent updates on its own data points. Between stages, the threads access to other portions of the data set. We tested the FFT algorithm with partition size varying from 8x8 up to 128x128.

FFT has a different access pattern from LU factorization in that an assigned home node is not the only one that would modify its partition data. This leads to more costly write misses and requires more invalidation operations in order to maintain coherence of cached partitions.

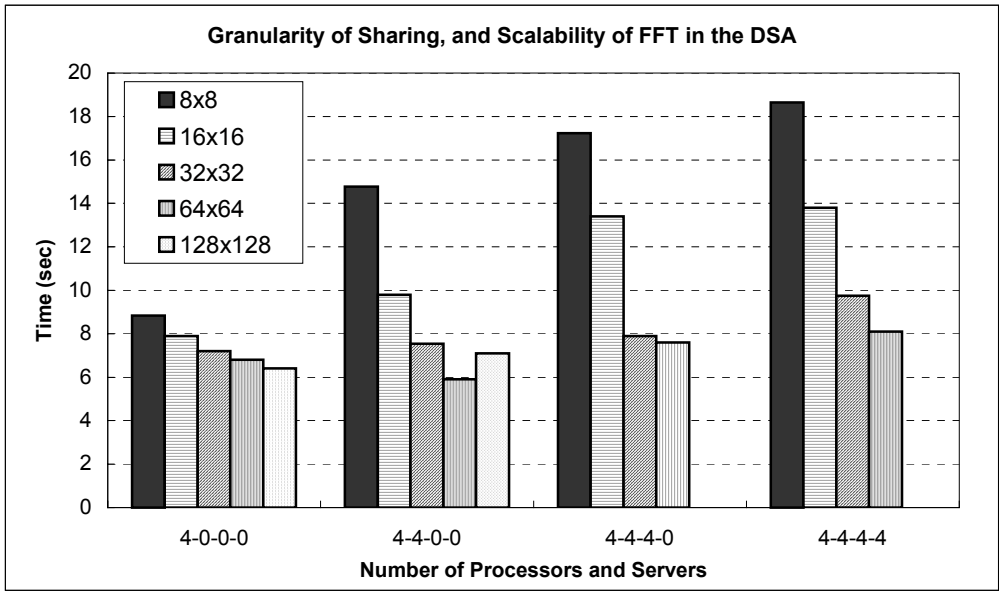

Figure 14: FFT for 1024x1024 double arrays on different virtual machine configurations.

Figure14 presents the performance of a 1024-point FFT with different partition sizes under the DSA support. With the expansion of the DSA virtual machine, the system incurs more overhead for ensuring cache coherence and results in more costly read and write misses. The ratio of computation to communication increases as the partition size increases. When the ratio goes up to a point where the partition size is of 64x64, the FFT application starts to observe slight performance gains when the virtual machine is expanded from a single node to a two-node system. But as the partition size goes up to 128x128, the communication cost becomes exceeding the advantage of having large computation segments and consequently leads to a loss of overall performance. Due to the limitation of the problem size, the maximum number of worker threads that the system can be configured in the case of partition size of 128x128 is limited to be eight.

Finally, we note that as in the case of LU factorization application, it is hardly seen performance gains with the FFT algorithm. It is mainly due to the high cost for remote data access in the cluster

environment. However, Figure 14 shows significant impacts of the coherence granularity on the overall performance. The DSA system benefits user-defined coherence granularity in the form of partition sizes.

5 Related Work

The DSA system tends to combine the better features of message passing and share address spacing models in clusters of SMP servers. It shares similar objectives as Global Array [21] and Split-C [8] to combine the portability and efficiency of message passing and ease of share address space for fairly large class of array-based scientific and engineering applications. Both the Global Array and Split-C were developed for LAN based clusters. Their virtual machines can neither be remotely installed nor reconfigured on the Internet. OpenMP [6] is a recently standardized directive-based API used with Fortran and C/C++ for programming shared address space machines in both SMP and NUMA architectures. Like the DSA programming model, OpenMP directives allow programmers to control data distribution for parallel efficiency. Its implementation on clusters of workstations relies upon software-based DSM systems; see [4][14] for examples. ParADE [17] is an OpenMP-based programming environment for SMP clusters and provides a hybrid execution model of message passing and shared address space. Its runtime system augmented the conventional software DSM systems with explicit message-passing primitives to reduce the synchronization overhead. The DSA system joins recent group of researches toward Java based parallel programming environment. MpiJava [2] and JPVM [11] suggested interface wrappers around MPI and PVM to support Java. Systems like Java/DSM [37], Jackal [30], and JESSICA [38] on the other hand suggested an approach of modifying Java virtual machine and/or Java compiler to support multithreaded programming in clusters.

Java/DSM [37] supports a Java-compliant shared address space programming model by implementing a modified Java virtual machine on top of the TreadMarks [1]. The granularity of sharing is a memory page. Page sharing is more appropriate for C language applications, where the programmer has explicit control over memory allocation. In Java the programmer has no control of which objects get allocated on each page. The DSA has different notation for granularity. It sets the partition size of which caching, replication and coherence protocol manage as the granularity level. With varying granularity levels, the programmers have more control over data distribution and alignment of the shared array for load balancing. Thus the application program has the ability to control the factor of different computation to communication ratio and thus control over the performance and optimization as seen fit. Jackal is a compiler-driven fine-grained DSM implementation of Java [22][30]. Its goal is to run unmodified multithreaded Java program efficiently on a cluster of workstations. It relies upon extensive compiler analysis and optimization in combination with supporting runtime optimization, including a self-invalidation based, multiple-write cache coherence protocol for shared data regions. Its goal is ambitious, but performance is yet to see. JESSICA[38] provides single system image illusion to Java applications via an embedded global object space layer. It implements a cluster-aware Java Just-in-time compiler to support transparent Java thread migration. In comparison, DSA supports dynamic reconfiguration of distributed virtual machine, which complements the existing thread migration work.

JavaParty [25] and other work on wide-area Java parallel computing systems like Manta [29][31] attempt to transparently add remote objects to Java and avoid the disadvantages of explicit socket communication. They extend the language by allowing a remote class to be hidden from local nodes. In

JavaParty [25], objects stay in their original processor and maintain stationary ownerships. They also observe a strong consistency between shared object read and write to simplify parallel programming. Manta [29][31] uses an optimized RMI communication protocol and inlined serialization routines for high performance. The optimized protocol also contains support for active messages in local area networks and TCP/IP messages in wide area computing. The DSA compensates remote access cost by allowing replication of array partitions as cached elements to the local nodes and maintains a relaxed consistency model for shared partitions.

Efforts to providing distributed shared objects for general-purpose distributed programming on the Internet can also be seen in JavaNow (Java for Networks of Workstations) [28], Sun Microsystems's JavaSpaces [15], and IBM's T-Spaces [33]. They provide global shared constructs, which allows processes to communicate with each other via shared proxy space. They are essentially extensions of the tuple-space concept in Linda [5] to the Internet. This architecture introduces costs associated with copying data and attribute matching for every remote access of objects. Object sharing in J-InterWeave [7] is realized by requesting an object with an URL-like identifier of its corresponding segment block. A machine-independent intermediate format was introduced to translate the data structures of objects across a heterogeneous shared memory environment.

6 Conclusions

In this paper, we have presented a Distributed Shared Array (DSA) runtime system to support Java multi-threaded programming on clusters of SMPs. As a hybrid of message passing and shared address space programming models, the DSA programming model allows programmers to explicitly control data distribution so as to take advantage of the deep memory hierarchy of the cluster architecture, while relieving them from error-prone orchestration of communication and synchronization at run-time. The DSA system features a directory-based cache coherence protocol in support of user-defined sharing granularity and a scheduling and communication proxy mechanism for reducing network contention. The DSA system is also 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 FFT 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. In particular, the experimental results have shown the DSA system benefits from user-defined coherence granularity.

One of the distinguishing characteristics of the DSA system is its mobility. Reconfiguration of the virtual machine is realized by migrating the DsaAgent daemon among physical servers. This mobility property greatly facilitates the load balancing and fault-tolerance for wide-area parallel computing. The details of the migration mechanism are described in [12]. Next, we will look into the migration decision problem that is the optimal timing to perform a migration operation. Other work that deserve further studies include optimization of low level communication mechanisms to reduce remote access latency and comprehensive performance evaluations with more real applications.

Acknowledgements

This material is based upon work supported in part by the U.S. National Science Foundation under Grants CCR-9988266 and ACI-0203592. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. NSF. A preliminary version of this paper appeared in the Proceedings of the 11th International Conference on Parallel and Distributed Computing and Systems [36].

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 Proc. of the First UK Workshop on Java for High Performance Network Computing, Europar* 1998.

[3] H. E. Bal, R. Bhoedigang, B. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, M. F. Kassehoek. Orca: A portable user-level shared object system. Technical Report IR-408, Vrije Universiteit, Amsterdam, June 1996.

[4] A. Basumallik, S.-J. Min, and R. Eigenmann. Towards OpenMP execution on software distributed shared memory systems. *In Proc. of Int'l Workshop on OpenMP: Experiences and Implementations*, May 2002.

[5] N. Carriero and D. Gelernter. Linda in Context. *Communication of ACM*, Vol. 32, No. 4, April 1989, pages 444—458.

[6] R. Chandra, et al, *Parallel Programming in OpenMP*, Morgan Kaufmann, 2001.

[7] D. Chen, C. Tang, S. Dwarkadas, and M. L. Scott. JVM for a Heterogeneous Shared Memory System. *In Proc. Of the 2nd Workshop on Caching, Coherence, and Consistency (WC3 '02)*, 2002.

[8] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Ericken, and K. Yelick. Parallel programming in Split-C. *In Proc. of SC'93*, pages 262—273.

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

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

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

[12] S. Fu and C.-Z. Xu. M-DSA: A mobile distributed shared array system. Technical report, Department of Electrical and Computer Engineering, Wayne State University, 2003.

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

[14] Y. Hu and H. Lu, A. Cox and W. Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, Vol.60(12):1512-1530, 2000.

[15] JavaSpaces. http://www.java.sun.com/products/javaspaces/specs/js.pdf

[16] H. Jiang and V. Chaudhary. MigThread: Thread migration in DSM systems. *In Proc. of Int'l Conference on Parallel Processing Workshops*, 581-588. August 2002.

[17] Y.-S. Kee, J.-S. Kim and S. Ha. ParADE: An OpenMP Programming Environment for SMP Cluster Systems. *In Proc. of ACM/IEEE Supercomputing Conference*, Nov. 2003.

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

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

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

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

[22] J. Pang, W. Fokkink, R. Hofman, R. Velderna. Model checking a cache coherence protocol for a Java DSM implementation. *In Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'2003)*, Nice, France, 2003.

[23] 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 IEEE International Conferrence on Cluster Computing (Cluster2000),* Nov. 2000.

[24] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technology*, pages 63—79, summer 1996.

[25] M. Philippsen and M. Zenger, "JavaParty - Transparent Remote Objects in Java", University of Karlsruhe, Germany. *Proceedings of PPoPP, Workshop on Java for Science and Engineering Computation*, July 22, 1997.

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

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

[28] G. Thiruvathukal, P. Dickens, S. Bhatti. Java on networks of workstations (JavaNOW): A parallel computing framework inspired by Linda and the Message Passing Interface (MPI). *Concurrency: Practice and Experience*, Vol.12(11): 1093—1116, November 2000.

[29] R. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-area parallel computing using the remote method invocation model. *Concurrency: Practice and Experience*, Vol.12(8):643—666, June 1999.

[30] R. Veldema, R. Bhoedjang and H. Bal. Jackal: A compiler based implementation of Java for clusters of workstations. *In Proc.of the 8th ACM SIGPLAN Symp. on Principles and Parallel Programming (PPoPP)*, Snowbird, June 2001.

[31] R.Veldema, R. Hofman, R. Bhoedjang, and H. Bal. Runtime optimizations for a Java DSM implementation. *Concurrency & Computation: Practice and Experience*. 15(3-5): 299-316, 2003.

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

[33] C. Wyckoff, T Spaces, IBM Systems Journal, vol. 37(3), 1998.

[34] C.-Z. Xu and F. Lau, *Load Balancing in Parallel Computers*. Kluwer Academic Publishers, 1997.

[35] C.-Z. Xu, and B. Wims. A mobile agent based push methodology for global parallel computing. *Concurrency: Practice and Experience*, Vol.12(8): 705-726, 2000.

[36] C.-Z. Xu, W. Brian, and R. Basharahil. Distributed shared array: an integration of message passing and multithreading on SMP clusters. *In Proc. of the 11th IASTED Int'l Conf. on Parallel and Distributed Computing and Systems,* Cambridge, MA, 305-310, Nov., 1999.

[37] 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.

[38] W. Zhu, C.-L. Wang, and F. Lau. JESSICA2: A distributed Java virtual machine with transparent thread migration support. *In Proc. of the 4th IEEE Conf. on Cluster Computing*, Sept 2002, pages 381-388.