

Implementation of a Portable Software DSM in Java

Yukihiko Sohda^{*}
Tokyo Institute of Technology
Tokyo, Japan
sohda@is.titech.ac.jp

Hidemoto Nakada
Electrotechnical Laboratory
Tsukuba, Japan
nakada@etl.go.jp

Satoshi Matsuoka[†]
Hirotaka Ogawa
Tokyo Institute of Technology
Tokyo, Japan
{matsu,ogawa}@is.titech.ac.jp

ABSTRACT

Rapid commoditization of advanced hardware and progress of networking technology is now making wide area high-performance computing a.k.a. the ‘Grid’ Computing a reality. Since a Grid will consist of vastly heterogeneous sets of compute nodes, especially commodity clusters, some have articulated the use of Java as a suitable technology to satisfy portability across different machines. Since Java’s natural model of parallelism is shared memory multithreading, one will have to support distributed shared memory (DSM) in a portable manner; however, none of the previous work on implementing Java on DSM has been a portable solution. Instead, we propose a software architecture whose goal is to achieve portability of DSM implementations across different commodity clustering platforms, while restricting the programming model somewhat, and implemented a prototype system, JDSM. Benchmark results show that the current implementation on Java incurs increased memory coherency maintenance cost compared to C-based DSMs, thus limiting scalability to some degree, and we are currently working on a solution to alleviate this cost.

1. INTRODUCTION

Rapid commoditization of advanced hardware and progress of networking technology is now making wide area high-performance computing, or so-called the ‘Grid’ Computing, a reality. Since a Grid will consist of vastly heterogeneous sets of compute nodes, especially commodity clusters, platform portability which enables programs to be downloaded from networks and to be executed is required. In addition, performance portability which provides good performance irrespective of platforms, is also needed. To satisfy portability across different machines, a language which has full code level portability and runtime optimization technologies such as JIT compiler would be suitable; needless to say, Java qualifies as such a language.

However, a Java Virtual Machine (JVM) implementation assumes

^{*}Research Fellow of the Japan Society for the Promotion of Science

[†]Also with Japan Science and Technology Corporation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2001 ACM 0-89791-88-6/97/05 ...\$5.00.

an underlying shared memory machine. As such, a multithreaded Java program cannot directly run on distributed memory parallel computers such as PC clusters. To facilitate such machines the distributed shared memory (DSM) functionality must be integrated into a JVM in some way. Previous work [2, 3, 15, 24] modified the JVM for such integration, sacrificing general platform portability of Java. Not only that the such customized JVMs will not be generally available, but also JVM modifications make it very difficult to employ existing JVM resources for high-performance computing, such as JIT compilers and efficient garbage collection algorithms, within such customized JVMs¹. As a result, previous DSM systems are not entirely appropriate for portable high-performance computing.

Instead, we propose a design of a portable Java DSM system called *JSDM* which could readily be run on top of a variety of JVMs, without their customizations. We employ a layered architecture where various layers of the DSM system can be tailored not only for portability, but also for achieving the best performance for a given platform, such as MPPs, PC Clusters, or a network of SMPs. In the lowest layer, a variety of low-level communication substrate could be employed with a common communication abstraction, from tightly-coupled, high-bandwidth low-latency messaging, to wide-area, secure communication over the Grid. In the higher layer, we can flexibly adapt to different architectures by allowing different memory consistency protocols and algorithms, depending on the characteristics of the application and the lower-level messaging layer. JavaDSM is also ‘Grid Aware’, in that programs and data files can be sent over a secure link with full authentication over a firewall, to a cluster with private addresses.

To accommodate such portability, JDSM requires that the users conform to a certain multithreaded programming style. This is in contrast to work such as cJVM[3] where the objective is to execute arbitrary multithreaded Java programs on a distributed memory machine. This in a sense is a technical tradeoff, and we will later show that for SPMD-style parallel programs, the restriction actually fits naturally with the style of such programs.

We evaluated JDSM on a PC cluster with 3 different JDKs and networks substrates of different speeds. The experimental results show that the JDSM generally scales well on different platforms, demonstrating the viability of portable implementation. On the other hand, we do observe that there are some overhead incurred in the synchronization, resulting in load imbalances and increased execution time. By improving on the area, we achieve speedups comparable to C-based DSMs, with execution time that is competitive.

¹In fact, as far as we know, none of the previous proposals employed a working JIT compiler.

2. DESIGN ISSUES OF PORTABLE JAVA DSMS

2.1 Major Design Issues

Because of the restriction as imposed by the Java Language Specification (JLS)[8, 14], a software DSM in Java is restricted from employing some of the techniques used in C-based DSMS. Below we discuss the design issues pertaining to the memory management and programming models in JDSM when considering the JLS:

2.1.1 Memory Management

Firstly, for DSM memory management, two technical choices must be made, 1) the units of memory management, and 2) the memory consistency model.

As memory management unit, a DSM can be either a) Page-Based, or b) Object-Based. C-based DSM systems often employ Page-Based management units for various reasons, including the availability of hardware paging to detect memory accesses with no program modifications and very low execution overhead. For Java, however, to employ Page-Based units the DSM memory manager must be able to access any memory within the JVM—this is possible when one modifies the underlying memory management of the JVM, as is with Java/DSM[24]. But in general, for a portable implementation such assumption cannot be made, and physical pages are completely hidden from the programmer within Java. Moreover, one requires an extremely sophisticated algorithm to efficiently cope with object copying for modern, incremental copying GCs employed in almost all high-performance JVMs such as the Sun HotSpot. On the other hand, Object-based management fits well with the underlying memory management of the JVM. Object references can be directly used, and even when objects are copied the JVM takes care of pointer forwarding etc. As such the Object-Based management does not require JVM modification, and is well-suited for portable DSM implementation in Java. The drawback is that object accesses must be checked in software, possibly causing non-negligible overhead.

Although there have been numerous proposals of memory consistency models, exactly which model is suitable for DSM on Java is not clear, since in general, the best consistency model depends on the memory access patterns of applications, and the properties of execution environments. In fact the general memory model of Java itself is in somewhat of a flux at the present time. Instead of tying the system down to a particular memory consistency model, it could be favorable to design a framework where it is easy to implement and customize different memory consistency models and protocols. For example, one could choose Invalidate/Update protocols, and efficient algorithms for depending on clustering environments such as Sequential Consistency or Lazy Release Consistency, etc.

2.1.2 Programming Model

In a very loose sense, the target of any Java DSM system is a parallel program written with Java threads. Systems could have varying degrees of *single system image* it presents as a programming model. On one extreme, a full illusion of a single JVM could be provided; one could relax this property for performance and portability, as will be described later.

To make such programs run on distributed memory machines without JVM modification, some form of program translation to employ DSM features might be required. Such translations involve analysis of which objects will be shared among the nodes, how the objects will be accessed, the relationship between threads on different nodes, etc. Although efforts have been made in the static analysis community, such analysis for general multithreaded Java

programs is difficult, and moreover, does not always result in an efficient parallelization of programs. One could do away with such analysis and handle the DSM operations entirely during runtime as is with cJVM, but one could sacrifice performance.

For JDSM, we restrict ourselves to mostly SPMD-style high-performance computing programs, where suitable DSM algorithms are known to perform efficiently in a clustering environment. We design the classes and their interfaces so that a SPMD-style programs can easily be written and existing ones ported onto JDSM with high performance. Moreover, since the programming model is similar to that of OpenMP, JDSM could possibly be used as an underlying implementation vehicle for Java version of OpenMP such as JOMP[11] on clusters.

For Object-Based management, access to shared memory can be done in two ways: one is the master-proxy scheme, where a proxy object with the same interface as the ‘real’ object to be shared is created. The proxy effectively hides the migration of objects it represents, and localizes the memory access checks to be performed on each node that holds the proxy. This scheme has the advantage that the complexity of program transformation is greatly reduced, involving very small code increase. There are several drawbacks regarding performance and storage, however; a) it cannot batch together successive memory accesses to multiple object fields for overhead elimination, b) the proxy class becomes a separate class, leading to some programming complications, and c) all field access involve method invocation, including array elements, causing considerable overhead. The alternative scheme is to insert memory access checks before and after object field accesses. In the worst case, memory checks will be inserted before and after *every* field accesses, but with simple optimizations one could effectively batch together such checks for efficiency.

For portable implementation, one has to cope with the fact that *different* JVMs are invoked on each node, and these JVMs have no knowledge that they are effectively sharing the heap with DSM. This causes some difficulties such as hash values. For arbitrary multithreaded programs spanning distributed memory, it is difficult to determine, say, the equality of objects using hash values due to differing hash values of objects amongst different JVMs. For JDSM, we alleviate this problem somewhat by restricting programs to be SPMD, and as will be described later, limiting how the sharable objects will be allocated. Due to this restriction, all the shared objects will be created in the same order, and the equality of objects can be determined by using the ordinal number as the key.

2.1.3 Coping with the ‘Grid’

For wide-area high-performance environment, e.g., the Grid, the following points must be considered as well. Firstly, a cluster system in a Grid must authenticate and authorize users in some way, and the DSM system must be executed under such (global) user ID. Not only data but also the program must be transferred in a secure manner to the cluster acting as the computing resource on the Grid, and adapted according to the particular environment of the cluster (e.g., the number of nodes, the communication substrate, the JVM being used, etc.). Since the nodes are shared among multiple users, Grid resource scheduling must coordinate with the resource acquisition of the JDSM.

2.1.4 Other Implementation Issues

In addition to the major design issues, we list below some other issues that must be considered for effective portability: specification of shared objects, object copying (migration), method invocations for shared objects, and handling of array objects and primitive types.

- The specification of objects to be shared must be made easy for the user. On the other hand, the extreme case i.e., allowing all the objects allocated in a program to be sharable, will be inefficient. JDSM only allows sharing of objects that have been explicitly allocated as sharable in the initialization phase of the program. Although this is somewhat restrictive, for SPMD programs ‘global’ data accessible from all the threads are usually relatively stable. The alternative is to perform static analysis to determine which objects are public and private, but this could be difficult/conservative. We are currently working to allow a restricted set of dynamically allocated objects to be shared to cope with OpenMP semantics.
- To copy or migrate an object between nodes, the object serialization is used. But user defined serializers are required for unserializable objects.
- Sharing a large object such as an array could result in substantial false sharing. Proper data distribution is necessary to avoid false sharing, and standard distribution methods as well as possibly array alignment must be supported.
- To share instances of primitive types (scalars), one could employ the wrapper classes, while the alternative is to employ different schemes for primitive type sharing. The latter could be more efficient, but would make the resulting system complex.

2.2 Summary of JDSM Implementation Choices/Policies

Based on the design issues, we derive the following design for JDSM. Although we have already mentioned or hinted at the design choices made, we nevertheless will summarize the choices below.

Units of memory management The unit of memory management unit is a Java object. We provide a framework from which various memory consistency models and protocols (e.g. Strict Consistency, Lazy Release Consistency, etc.), can be derived. Currently, JDSM defaults to Lazy Release Consistency[12] with Write Invalidate.

Programming Model User programs are SPMD-style Java multi-threaded programs. Access checks, in the form of method invocation to memory consistency operations, are inserted manually by the user or a higher-level program translator for field read/write accesses. Method invocations are treated as write accesses with no special checks. Shared objects are registered at object creation time during initialization phase.

Coping with the Grid In order to achieve widespread portability, we currently employ ssh for wide-area authentication, although other infrastructures such as Globus[7] could be used. Ssh establishes a communication stream between the server (i.e., the remote host) and the client. For easing the management of server programs, the client initially invokes a Java management process called ClusterManager; the ClusterManager subsequently invokes a server process on each cluster node. The user program and data are securely transmitted using a ssh stream from the Client, possibly across firewalls. Here, class files are loaded from the ssh stream on the (remote) server host using a customized class loader.

Other Implementation Issues As mentioned earlier, the objects that are shared can only be declared during program initialization, while objects created inside parallel execution threads

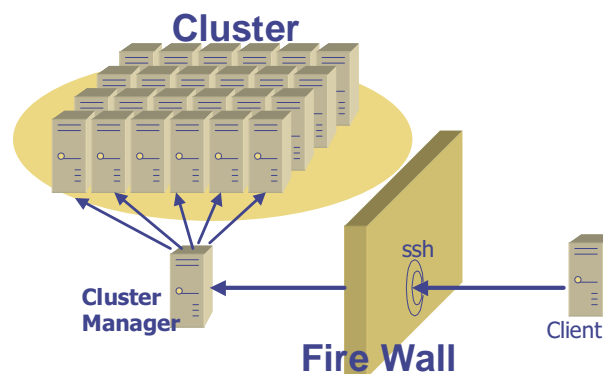


Figure 1: JDSM Overview and execution image

are private to the thread. Object migration uses the standard Java serialization, and as such unserializable objects are not supported. By using a standard serializer, programs can identify objects (e.g. equal), but must be careful about their overhead. Array objects are shared with proper distribution, default being block distribution per size. Primitive type values are not supported for simplicity, requiring wrapper classes.

3. JDSM IMPLEMENTATION

We now describe the JDSM implementation in detail. The JDSM system has a modularized and layered architecture for portability, and is also architected as a class framework. Roughly speaking, the JDSM system is composed of three modules (Figure 1).

ClusterManager The module for managing the DSM Servers on a cluster. Basically, the ClusterManager invokes computation threads on each Server in response to a Client job invocation request.

Server The Server runs on each cluster node, and executes user programs on behalf of the client. Server is invoked by ClusterManager, then mutually establishes connections with Servers on other nodes.

Client The Client sends the user’s computation requests to the ClusterManager. During computation, the Client acts as a file server for sending programs and data to the Servers. As seen in Figure 1, the Client can communicate with the ClusterManager and Server nodes across a firewall in a limited fashion, leading to flexible and secure system in a Grid environment.

The Server itself is composed of three components that constitute a layered architecture (Figure 2):

SPMD layer Provides the SPMD programming model interface to which the user programs to. Note that this layer provides only the APIs—the actual DSM behavior is provided by the underlying DSM layer.

DSM layer The DSM layer performs DSM consistency checks and operations using the high level communication interface provided by the underlying Trans layer. The layer is constructed as a pluggable class framework so that various consistency algorithms and protocols can be implemented. By default, the system implements the Lazy Release Consistency which

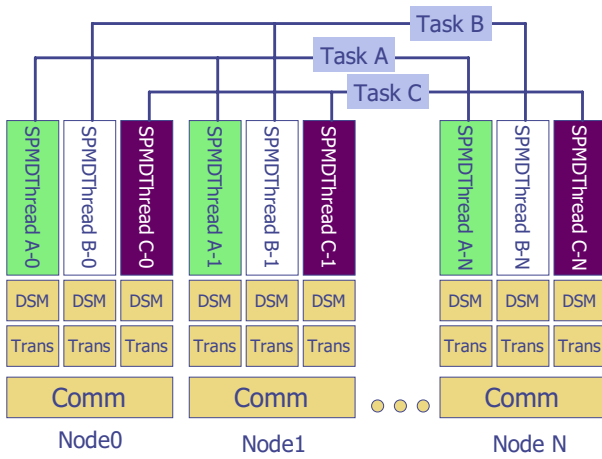


Figure 2: Component layer image on Server nodes.

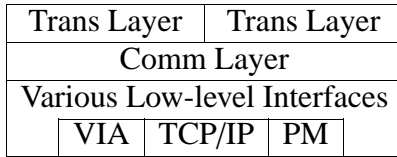


Figure 3: Communication Component Hierarchy

can be run with good scalability in a clustering environment without extremely fast networks. More aggressive protocols can also be employed for MPP and fast messaging environments.

Trans layer The Trans provides an abstract, high level communication layer that supports various DSM-specific operations as will be described. This layer only assumes ‘logical’ thread locations. Mappings to physical locations and actual communication happens in the lower-level Comm Layer.

Comm layer The Comm layer is the physical communication layer, and is shared by all SPMD threads. This layer is designed to be pluggable, and hides the differences of various physical networks equipped on clusters. Also, this is the only part of the system where native calls could be facilitated via JNI—the rest of the system are pure Java. By default a generic TCP/IP socket communication (which does not use JNI and is thus pure Java) is provided, but we are also developing modules for directly calling PM[20] and VIA[10] interfaces via JNI.

3.1 Trans & Comm Layers

We first describe the Trans & Comm Layers in detail. As mentioned above, a portable DSM system must be able to support a) a variety of (perhaps multiple) networks that interconnect the cluster nodes, and b) co-existence of multiple communication threads (Figure 3).

The standard JDK only supports TCP/IP socket communication between separate JVMs. To support different interconnects and their fast messaging protocols (such as VIA, PM, AM[22], GM, etc.), in a portable manner, the communication layer abstracts out the differences between the underlying communication substrates. When porting to a platform with a new network or a low-level protocol, one only needs to implement a new Comm layer module, and

method	description
asend	asynchronous message sends
received	message receive (callback method)
expressSend	send express messages

Table 1: Declared methods in Trans Class

method	description
init	initialization
fin	finalization
asend	sending messages
expressSend	send express messages

Table 2: Declared methods in Comm interface

the rest of the library nor the user program is affected. By all means, not only the difference in the performance of different communication substrates must be considered, but also the overhead of Java (such as the cost of JNI invocation) must be technically considered, which we will demonstrate in the benchmarking section.

The Trans layer ties together the computation threads with the Comm layer. It also abstracts out the physical node location of threads. It also provides interfaces such as the followings for implementing DSM behavior in the DSM layer (Table 1).

The asend method sends asynchronous messages to a thread, whether it is on a local node or a remote node. The received method is a callback method called on message receives. The received method in turn employs another thread to call some DSM layer method according to the message. Here, to avoid the cost of new thread creation, we employ the standard thread pooling technique. The expressSend method is used for sending a fixed-size, high-priority ‘express message’ employed in synchronizations such as barriers, and system control. Express messages have a special format, and treated on an out-of-band control path compared to standard messages, in which objects and classes are serialized.

The Comm layer is composed of the Comm interface (Table 2) for sending messages and a Dispatcher that receives messages and forwards them to an appropriate Trans layer methods. Comm itself does not provide an interface for receiving messages, as the message is routed to the Dispatcher on message reception. The Dispatcher in turn calls the appropriate received method of a Trans layer instance of the designated recipient. As such, one only needs to implement the message sending interface in Table 2 and a Dispatcher interface to cope with a new communication substrate.

3.1.1 SocketComm

As an example, we demonstrate the implementation of the standard SocketComm, which is written purely in Java. SocketComm employs Java Socket and ServerSocket classes, and establishes full TCP/IP peer-to-peer socket connections (i.e., $N * (N - 1)$ connections for N nodes) between the nodes. The socket must be read and written simultaneously; however, since Java does not have simultaneous read/write interface such as the Unix select, an extra thread is allocated to read messages from the stream. When a new message arrives, the thread checks the message tag, and calls either the received or the expressReceived method in the Dispatcher according to the tag.

Although SocketComm is pure Java, faster communication substrates will require parts of their implementation to be in native methods for fast communication and linking to the C library of the substrate. Currently, we have implemented the Comm interfaces for PM and VIA using JNI.

```

public abstract class SharedObject{
    Object obj; // shared object
    int key; // key(registration number)
    int myNode; // node number
    private SharedObjectStatus status;
        // status of shared object
    void lock(); // lock this shared object
    void unlock();// unlock this shared object
    ...
}

```

Figure 4: SharedObject abstract class

method	description
<code>init</code>	initialization
<code>fin</code>	finalization
<code>register</code>	register shared objects
<code>acquire</code>	get lock for write access to shared object
<code>release</code>	release lock
<code>update</code>	get latest shared object status
<code>acquireAsync</code>	asynchronous acquire
<code>updateAsync</code>	asynchronous update

Table 3: Declared methods in SharedObjectPool

3.2 DSM Runtime Implementation

As mentioned earlier, for the current implementation, the DSM is achieved by using Lazy Release Consistency where the Write Invalidate protocol is used. The basic algorithm is essentially same as that in C. The consistency protocol for the program is specified in the configuration file. The implementation essentially realizes a “Global Object Space”, because the unit of sharing is a Java object.

Each shared object is serialized by the Java serializer, then sent to other nodes, and as such the shared object has to implement `Serializable`. Since the Java serializer performance is low, we are in the process of implementing a faster serializer, for example, a primitive array specialized serializer.

Each shared object is wrapped and managed in the (subclass of) `SharedObject` abstract class (Figure 4). Subclasses include `SharedObjectOne` which manages single-instance objects, and `SharedObjectForBlock` which manages array objects with block distribution, etc.

DSM runtime in JDSM system defines an abstract class `SharedObjectPool`, and specific DSM algorithms and protocols will subclass and define the required interfaces as shown in Table 3 to implement their respective functionality. The algorithms must also utilize the Trans layer for inter-node communication, and resorting to other means of back-door communication is not allowed.

Below, we describe the interface of the `LazyRelease` class which inherits from `SharedObjectPool`:

init method Initializes the management tables of shared objects, acquires information on the execution environment (number of nodes, etc.) from `Comm`, creates a thread to process request from other nodes.

register method Registers a shared object by wrapping it in the (subclass of) the `SharedObject` abstract class, and makes an entry in the shared object management table. For algorithms that allow owner migration, the current implementation sets the owner node to be node rank 0.

update method Called with the object to be updated as an argument, and updates the local node view of the object to be up-to-date.

```

public abstract class Spmd{
    public Comm comm;
    public SharedObjectPool pool;
    public void init();
    public void start();
    public void fin();
}

```

Figure 5: Spmd abstract class

acquire method The object to which the write lock should be acquired is called as an argument, and an exclusive access right to the object is granted. Notification (invalidation in the case of write-invalidate) message is sent to all the nodes which have local copies of the object.

release method The object to which an exclusive access is being held is called as an argument, and the lock is released, notifying all relevant nodes.

updateAsync method For efficiency, JDSM offers asynchronous consistency management. In contrast to `update`, where the returned argument is the original object to be updated in the call and the update will have occurred on return, for `updateAsync` a `Future` object is immediately returned. The actual updated target shared object is obtained by calling the `touch` method, which blocks until the update is actually complete. A proper use of method allows update latency to be effectively hidden (although Lazy Release Consistency we implement already alleviates some of the latency), an important optimization in clusters with higher communication latency.

acquireAsync method Similar to `updateAsync` in that it facilitates asynchronous acquire with `Future` objects.

fin method Finalizes the DSM program by discarding the distributed shared objects, management tables, etc.

3.3 User Program

The target user program is a SPMD-style multithreaded program. The user programs his code by subclassing the `Spmd` abstract class (Figure 5).

The `init` and `fin` methods are executed sequentially and exactly once, at the beginning and at the end of program execution respectively. The user must allocate and register the shared objects in the `init` method using the `register` method, and finalize them in the `fin` method. The user programs the main body of his parallel code in the `start` method. Here, the number of threads appropriate for each node (usually matching the number of CPUs in the node, although it can be specified otherwise) is fired up, and will all call the `start` method in parallel. All objects allocated within `start` will be private to that node, although we are considering limited relaxation of this to accommodate Java OpenMP.

Here, Figure 6 is a sample JDSM program. Multiple threads spanning across the cluster shares three vectors, and multiply the vectors at will in a consistent manner. First, three arrays of double are declared and registered in `init` (1). Then threads starts executing in parallel. Each thread updates access reads to the array b and c (2), acquires access writes to the array a (3), stores the multiplication result and releases it (4).

3.3.1 Overall Setup and Workflow of DSM Execution

Given such a client API, the underlying JDSM system executes the user program in the following way:

1. The `ClusterManager` is initialized and started,

```

public class VecMul extends Spmd{
    public double[] a,b,c = new double[length];
    public int length;
    public void init(){
    ...
        pool.register(a, lengthPerNode);           // (1)
        pool.register(b, lengthPerNode);
        pool.register(c, lengthPerNode);
    }
    public void start(){
        for (int i = lengthPerNode * myThread ;
            i < lengthPerNode * (myThread + 1) ; i++){
            pool.update(b, i);           // (2)
            pool.update(c, i);
            pool.acquire(a, i);         // (3)
            a[i] = b[i] * c[i];
            pool.release(a, i);         // (4)
        }
    }
    public void fin(){ ... }
}

```

Figure 6: User program example

2. Servers are started by the ClusterManager on each node,
3. Communication is established between each node, the Comm layer is initialized,
4. The user program is executed by the Client, possibly outside the firewall; secure communication established between the Client and the ClusterManager,
5. DSM runtime and Trans layers are initialized, class files are securely shipped to servers from the client using secure streams (ssh port forwarding in the current implementation).
6. Execute `Spmd.init` on each node,
7. Execute `Spmd.start` for specified number of threads on each node, I/O requests are forwarded and processed by the Client,
8. Execute `Spmd.fin` on each node
9. Finalization of DSM runtime and Trans layer on each node

4. PERFORMANCE EVALUATION

We have tested the performance of JDSM system by using a) different JVMs and JIT compilers, as well as b) different underlying communication substrates (interconnects and protocols). The benchmark is a Java ported version of the LU Kernel and Water from SPLASH2[23].

4.1 Evaluation Environment

The evaluation environment is the Presto PC Cluster, which features 64 500MHz Celeron PC nodes with 512MB memory (of which 32 is currently used for benchmarking, because only 32-nodes are facilitated with Myrinet), with multiple interconnects (multiple Fast Ethernet links and Myrinet). Operating system is Linux 2.2.16 with RWCP SCore[9] 3.2 extensions. The JDKs subject to evaluation are IBM JDK1.3, Sun JDK1.3.0 ('Hotspot VM'), and Sun JDK1.2.2('classic VM')+OpenJIT-1.1.15.

4.2 Basic Performance

As a basic test, we evaluated the performance of the Comm modules and the JDSM runtime.

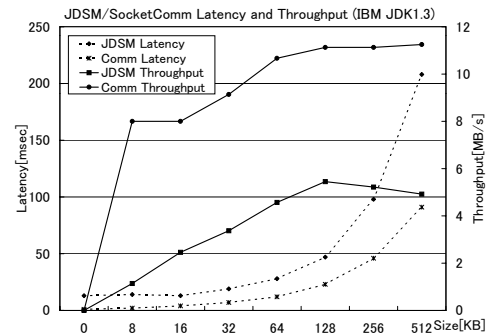


Figure 7: JDSM Runtime Basic Performance (SocketComm)

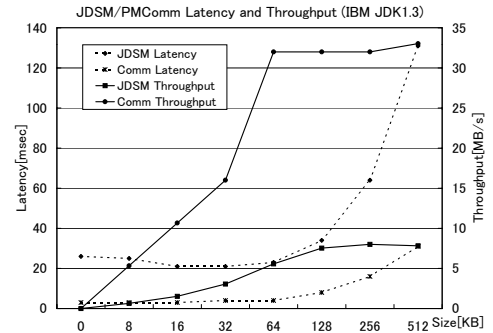


Figure 8: JDSM Runtime Basic Performance (JPMComm)

4.2.1 SocketComm

Table 4 shows the one-way latencies of native socket and Java socket in SocketComm for varying data sizes. Compared to native sockets, we see that Java SocketComm incurs approximately 1.3 times latency overhead, and saturates around 9.3MB/s which is slightly lower than the saturated native socket bandwidth. This shows that the overhead imposed by Comm layer is reasonably small for relatively slow socket communication.

4.2.2 JPMComm

We next compare the performance of native PM and its Java interface we have developed, called JPMComm (Figure 5). PM[20] is being developed at the Real-World Computing Partnership (RWCP) of Tsukuba, Japan, and is known to be one of the fastest communication protocol on top of Myrinet. JPMComm provides Java JNI interface to the underlying low-level PM primitives for JDSM. Because of low latency/high-bandwidth characteristics of PM/Myrinet, Java overhead manifests itself quite easily compared to SocketComm. We employ caching as well as other techniques to lower the JNI overhead as much as possible. Still, for small sizes we almost double the latency (6.82 usec vs. 10.4 usec) while on larger data sizes the ratio goes down. Even for native PM, we see that performance is limited to only 34MB/s; this is far lower number than reported for other PM publications. We have not pinpointed the exact cause, but microbenchmarks exhibit problems in PCI hardware, in that the PCI DMA-write performance is somehow severely restricted on our particular PC.

4.2.3 JDSM Runtime Basic Performance

We next measure the basic JDSM runtime overhead. We employed the IBM JDK1.3 as the JVM, and SocketComm and JPMComm as the communication layer. In the PingPong benchmark, we have

Mess. Size(bytes)	Socket		SocketComm	
	Latency(usec)	Bandwidth(MB/s)	Latency(usec)	Bandwidth(MB/s)
1	86.68	0.011	109.7	0.009
16	89.00	0.179	111.3	0.143
256	155.4	1.646	177.4	1.442
4096	671.6	6.098	699.0	5.859
32768	3109.7	10.53	3496.5	9.372

Table 4: Latency and bandwidth on SocketComm and Java Socket

Size(bytes)	PM		JPMComm	
	Latency(usec)	Bandwidth(MB/s)	Latency(usec)	Bandwidth(MB/s)
1	6.82	0.147	10.4	0.096
16	7.02	2.266	10.5	1.523
256	18.00	14.25	24.0	10.66
4096	167.6	24.44	211.0	19.41
65536	1923.8	34.07	2087.3	31.39

Table 5: PM and JPMComm Latency and Bandwidth(Myrinet)

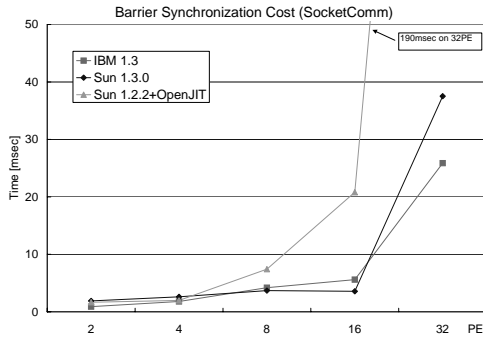


Figure 9: Barrier Synchronization Cost (SocketComm)

the two nodes share an array object, and alternatively perform writes to the array. The full DSM protocol is operational, including calls to `acquire`, `release`, as well as sending of invalidation messages. The result is compared with direct calls to the Comm layer (Figure 7, Figure 8).

Both Comm interfaces exhibit good performance, saturating the physical bandwidth—at 64KBytes, we observe 11MB/s for SocketComm and 32MB/s for JPMComm. On the other hand, when DSM is layered on top, we only obtain 5.4MB/s and 7.5MB/s for SocketComm and JPMComm at 128KB, respectively, only obtaining 1/2 to 1/4 of the peak bandwidth. This is primarily due to the overhead of object locks on `acquire`, as well as the serialization overhead of objects.

We also measure the cost of barrier synchronizations (Figure 9). Though at this time we adopt the shuffle exchange as a barrier synchronization algorithm, arbitrary algorithm can be used. We observe that the cost is relatively small for 16 processors, but it increases for 32 processors. We speculate that the slowdown is incurred by improper thread scheduling in the JVM, but we are still investigating this.

Such low-level benchmarks do not give the whole story; indeed the question is, will the loss in performance be negligible so that we obtain fast and scalable DSM system. We investigate this in our applications benchmarks.

Time[sec]	IBM JDK1.3	Sun JDK1.3	C
LU (2048)	68.4	90.6	58.0
Water (4096)	126.2	327.8	98.0

Table 6: SPLASH2 on an uniprocessor

4.3 Benchmarking with SPLASH2 LU and Water

We measure the performance of JDSM with SPLASH2 LU Kernel and Water ported to JDSM as an example scientific application. For benchmarking purposes, We have varied the matrix sizes from the original SPLASH2 LU kernel from 1024 by 1024 to 2048 by 2048. We also compared the performance with uniprocessor execution of the code for both C and the JDKs we have employed (Table 6). We see that Java is competitive with C, although not entirely equivalent. Comparing this uniprocessor performance with Figure 11 and Figure 13 for 1 node, we also see that the cost of DSM operations add less than 15% overhead to sequential code under JIT compilation.

For execution on clusters, Figure 10 shows the results for IBM JDK 1.3, Sun JDK 1.3.0 (HotSpotVM) and Sun JDK1.2.2 + OpenJIT-1.1.15, respectively. We observe that scalability is somewhat restricted — for 16 nodes we are obtaining only approximately factor of 2 speedup. Looking at the time breakdown reveals that the problem size is too small for the number of processors, and communication dominates the computation time. In fact, much of the communication time is barrier operation, and preliminary analysis shows that, on the current Linux platform, poor thread scheduling of native threads hinders the compute threads to be reactivated after barrier synchronization. As such, we need to find synchronization tricks to circumvent this problem. If we observe the LU Kernel core computation (Interior), we see that IBM JDK1.3 achieves 3.7 times while Sun JDK1.3.0 achieves 4.7 times speedups.

By doubling the problem size to 2048×2048 (computational complexity per node increase by a factor of 8), we obtain the results as shown in Figure 11. Here, we obtain speedups of approximately 5.8 to 7.0 times, and the interior speedup exceeds over a factor of 10 for 16 processors. The performance saturated with 16 processors as the performance for 32 processors is equal to for 16

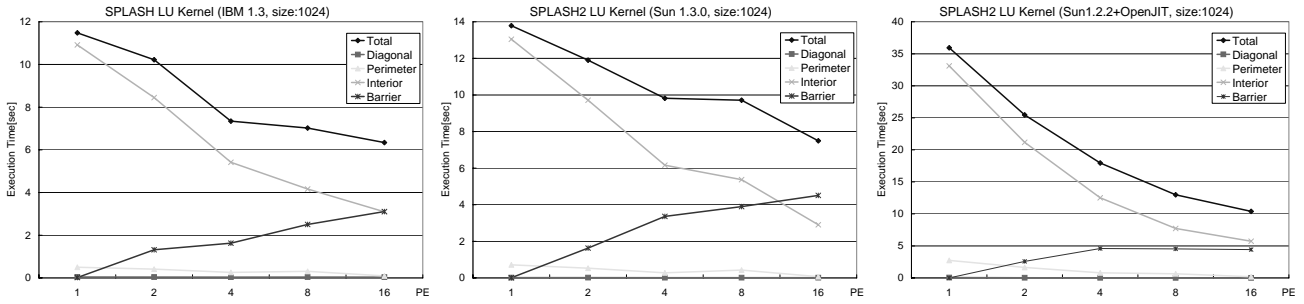


Figure 10: SPLASH2 LU Kernel (Matrix Size:1024) with SocketComm

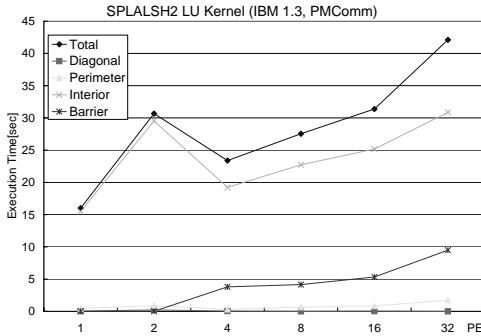


Figure 12: SPLASH2 LU Kernel (Matrix size:1024) on IBM JDK1.3. with PMComm

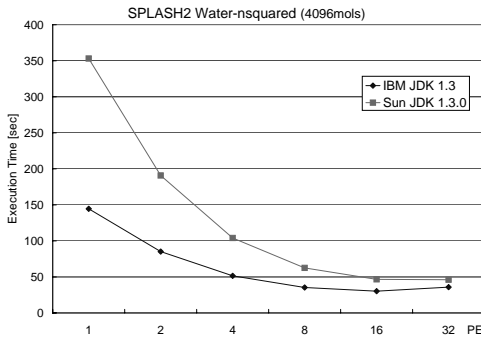


Figure 13: SPLASH2 Water-nsquared (4096mols) with SocketComm

processors. Since the speedup ratio increased as the problem size increased from 1024×1024 to 2048×2048 , further increase in speedup ratio is expected for realistic problem sizes.

As a reference, we attach the results of JPMComm (Figure 12). Since JPMComm is still in development stage, we see that the performance is unstable.

Finally, we show the measurement with the SPLASH 2 Water-nsquared benchmark with SocketComm (Figure 13). The number of molecules is 4096, the number of timesteps is 3. We obtain the peak speedups of approximately 4.5 to 7.5 times for 16 processors.

5. RELATED WORK

As a precursor research, we have demonstrated a similar portable DSM scheme with the OMPC++[19] system. OMPC++ translates multithreaded SPMD C++ programs onto a software DSM imple-

mented with a MPC++ fine-grained object-based message-passing language using the program transformation feature of OpenC++[5]. Portability of OMPC++ relies on MPC++, i.e., MPC++ is the target language to compile the DSM code to. Although similar, it did not feature pluggable DSM protocols and communication layers, nor had to be concerned with technical restrictions of Java.

Although there are numerous DSM systems proposed for C since the inaugural work by Kai Li[13], research on DSM systems for Java have not been many, despite that Java's natural model of parallel programming is shared memory + threads:

Java/DSM[24] Java/DSM is an early Java DSM system, where it modifies the underlying memory management of the 'classic' JVM so that it uses an existing C-based DSM system TreadMarks[1]. This allows simple execution of Java on top of clusters if TreadMarks is available; on the other hand, because of substantial modification to the memory system, existing JIT compilers cannot be used. Being a page-based system, it is not clear if advanced copying garbage collectors on modern VMs could be easily combined with TreadMarks.

cJVM[3, 4] cJVM is a DSM system being developed by IBM Haifa. It achieves a single system image by modifying the JVM. Contrasting to Java/DSM, it is a standalone JVM where memory management unit is object-based, and remote object invocations as well as field accesses are handled with proxies. cJVM features an impressive array of functionalities to implement single system image, such as a thread model which supports a transparent method shipping, distributed class loading, and considerations for distributed I/O, etc. Such features are necessary because cJVM is targeted for traditional server-style applications—being able to replace, say, Java web server on SMPs with a clustered version. On the other hand, as far as we know it does not facilitate a JIT compiler due to various changes within the JVM including memory management, object layout, and additional bytecodes. Moreover, although no scientific benchmarks have been run, we speculate that for such applications master-proxy model will turn out to be slow. Finally, portability is a problem, as it only runs on its customized JVM.

JESSICA[15] JESSICA is similar to cJVM in that the JVM is modified to achieve single system image on a PC cluster. The difference from cJVM is the support of load balancing via thread migration, and the reliance on existing DSM system for distributed memory management, as is with Java/DSM. Again, performance and portability would become issues in a wide-area, high-performance environment.

Hyperion[2] Hyperion translates Java into C code, and then employs existing DSM libraries and system threads to imple-

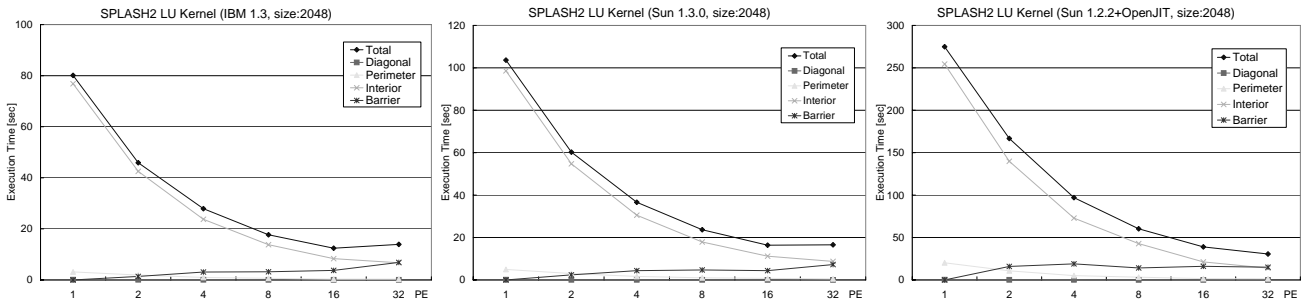


Figure 11: SPLASH2 LU Kernel (Matrix Size:2048) with SocketComm

ment DSM in Java. DSM-related operations are inserted at program transformation time. As is with JDSM, Hyperion features multiple memory consistency algorithms/protocols, as well as being able to employ various low-level communication layers. Portability does suffer due to the reliance on the libraries as well as not being able to support Java's features fully (such as dynamic program loading, security, etc.), but nevertheless it would be interesting to directly compare the performance with JDSM, since on scientific programs straightforward Java to C conversion could result in efficient code.

Jackal[21] is a high-performance Java DSM system where a highly efficient Java messaging runtime, Manta, is combined with an efficient static compiler. Although there are no direct comparisons with C, the early reports of Jackal performance seems to compare in par with the IBM JDK1.3.0 JIT. Jackal also performs various optimizations at compile time; it would be interesting to compare the performance of Jackal with that of JDSM for a common benchmark.

To summarize, all the systems suffer from assuming a customized Java execution infrastructure; one cannot use arbitrary JVMs as is possible with JDSM. Unless the underlying runtime supports it, it may not be easy for the user to customize his code to employ whatever the appropriate underlying communication substrate is available for the cluster, nor being able to customize the consistency protocols (except for Hyperion). Performance suffers due to the fundamental lack of JIT compilers due to JVM modifications, or straightforward translations, unless a custom compiler is employed as is with Jackal.

Compared to such systems, JDSM does offers portability across different JVMs as well as reasonable performance, exploiting existing Java infrastructure. JDSM programs has so far been run on 3 different JVMs with full JIT compiler support. The communication substrates implemented or in development include Sockets, VIA on Ethernet, VIA on Myrinet, and PM on Myrinet. Although there are some performance issues to be resolved, JDSM is competitive if not matching the performance achieved by C-based DSM systems.

The drawback of JDSM is that, it does not offer the single system image to arbitrary Java multithreaded programs. Programs have to be written in a certain SPMD style, and the nodes are used merely as compute engines where external communication e.g., file I/O are centralized and done remotely by the Client. For scientific computation, we feel that such a restriction is acceptable; for example, we have observed that SPLASH2 programs are written in a way such that they are readily portable to JDSM. On the other hand, JDSM is not appropriate for transactional or web style multithreaded programming. Although we plan on expanding the scope of programs

appropriate for JDSM, it is not clear whether such expansion will sacrifice the portability and/or performance as currently enjoyed by JDSM.

JavaParty[18] is a pure-Java system that allows transparent remote object method invocation and object migration. JavaParty shares the advantage with JDSM in that standard JVM platforms can be employed, and as such is portable across a variety of platforms. JavaParty relies on RMI for communication (and as a result is a proxy-based system), and in fact offers its own version of (pure Java) RMI to achieve higher performance, since the use of generic RMI would be excessively slow. JavaParty is well-suited for programming in the wide-area setting, as it features the full security functionality of RMI; on the other hand as JavaParty is not a pure DSM system, it is not clear how that would affect the programming style w.r.t. SPMD programs, or how it will perform due to its proxy-based design.

6. CONCLUSION AND FUTURE WORK

We proposed JDSM, a portable, Java DSM system for wide-area, high-performance computing. We discussed the design issues for designing such a system in Java, especially in the light of various restrictions imposed by JLS. We then presented a concrete design and implementation in the form of JDSM, describing how portability could be achieved at the expense of restricted programming model. Test runs and Benchmarks show that JDSM is portable across a variety of JVMs and execution platforms, and performance is generally reasonable, with needs for improvements in some specific cases.

JDSM currently is in a sense a run-time library for DSM support—we are currently developing a program translator to insert the necessary field read/write check operations automatically. For this we are currently investigating two possibilities; one is to use dynamic program editing systems such as Javassist[6] to customize the program offline or at load time. The other option is to employ JIT compilers such as OpenJIT[16, 17] to perform JIT compile-time customization. The former has the advantage of being able to adapt to a variety of JVMs, since the result is still a valid portable Java class file, while the latter could provide the full compiler infrastructure to optimize away the checks, etc. We could also use JDSM as a backend to Java OpenMP bindings such as JOMP[11], in which case the checks are inserted by the OpenMP frontend. As mentioned earlier, JDSM object allocation semantics must be modified to accommodate OpenMP, however.

There are other technical challenges that need to be considered for future work. We need to investigate the performance interactions between the lower-level communication substrate and different memory consistency algorithms and protocols, especially with respect to scalability. We need to port more SPLASH2 benchmarks

as well as other programs to test the scalability of JDSM. Moreover, we must strive to alleviate some of the performance problems as pointed out in the Benchmarking section, so that the system can be distributed and deployed in a real-life setting. Security needs to be enhanced, especially for allowing remote resources on Server nodes to be accessed. Joining of multiple clusters in a secure manner across high-bandwidth wide-area interconnect in the presence of firewalls and private addresses must be coped with.

7. REFERENCES

- [1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.
- [2] G. Antoniu, L. Bouge, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Implementing Java consistency using a generic, multithreaded DSM runtime system. In *Parallel and Distributed Processing. Proc. Intl Workshop on Java for Parallel and Distributed Computing*, volume 1800 of LNCS, May 2000.
- [3] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Cluster Aware JVM. In *Proceedings of International Conference on Parallel Processing '99*, pages 31–39, Jun. 1999.
- [4] Y. Aridor, M. Factor, A. Teperman, T. Eliam, and A. Schuster. A High Performance Cluster JVM Presenting a Pure Single System Image. In *Proceedings of ACM 2000 Java Grande Conference*, pages 168–176, June 2000.
- [5] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA '95*, pages 285–299, 1995.
- [6] S. Chiba. Javassist — A Reflection-based Programming Wizard for Java. In *Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, October 1998.
- [7] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *international Journal of Supercomputer Applications*, 1997.
- [8] J. Gosling, B. Joy, and G. Steel. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.
- [9] A. Hori, H. Tezuka, and Y. Ishikawa. An Implementation of Parallel Operation System for Clustered Commodity Computers. In *Cluster Computing Conference '97*, Mar. 1997.
- [10] <http://www.viarch.org/>. Virtual Interface Architecture Specification, Dec. 1997.
- [11] M. Kambites and J. Bull. JOMP – An OpenMP-like Interface for Java. In *ACM 2000 Java Grande Conference*, pages 44–53, Jun. 2000.
- [12] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Dept. of Computer Science, Rice University, Dec. 1994.
- [13] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov 1989.
- [14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.
- [15] J. M. M. Matchy, C. Wang, F. C. M. Lau, and X. Zhiwei. JESSICA: Java-Enabled Single-System-Image Computing Architecture. In *1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, June-July 1999.
- [16] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT — A Reflective Java JIT Compiler. In *Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, pages 16–20, Dec. 1998.
- [17] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compile Framework for Java. In *Proceedings of ECOOP '2000 - Object-Oriented Programming*, number 1850 in LNCS, pages 362–387, Jun. 2000.
- [18] M. Philippsen and M. Zenger. JavaParty — Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1125–1242, 1997.
- [19] Y. Sohda, H. Ogawa, and S. Matsuoka. OMPC++ — A Portable High-Performance Implementation of DSM using OpenC++ Reflection. In *Proceedings of Reflection '99, LNCS 1616 Meta-Level Architecture and Reflection*, pages 215–234, July 1999.
- [20] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High Performance Communication Library. In P. Sloot and B. Hertzberger, editors, *High-Performance Computing and Networking '97*, volume 1225, pages 708–717. Lecture Notes in Computer Science, Apr. 1997.
- [21] R. Veldema, R. Bhoedjang, R. Hofman, C. Jacobs, and H. Bal. Jackal: a compiler supported, fine grained Distributed Shared Memory implementation of Java. Technical report, Division of Mathematics and Computer Science, Vrije University, Amsterdam, 2000.
- [22] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.
- [23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Jun. 1995.
- [24] W. Yu and A. Cox. Java/DSM: a Platform for Heterogeneous Computing. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, volume 43.2, pages 65–78, Jun. 1997.