Java/DSM:
A Platform for Heterogeneous Computing

Weimin Yu and Alan Cox
Department of Computer Science
Rice University
Houston, TX 77251
{weimin, alc}@cs.rice.edu

July 31, 1997

Abstract

In this paper we describe a system for programming heterogeneous computing environments based upon Java and software Distributed Shared Memory (DSM). Compared with existing approaches for heterogeneous computing, our system transparently handles both the hardware differences and the distributed nature of the system. It is therefore much easier to program.

Java is a good candidate for heterogeneous programming because of its portability. Java provides the Remote Method Invocation (*RMI*) mechanism for distributed computing. However, our early experience with RMI showed that the programmer must spend significant effort on such problems as data replication and the optimization of the remote interface to improve communication efficiency. Furthermore, the need for reference marshaling is not completely eliminated by RMI's effort to facilitate the sharing of linked structures between machines.

A DSM system provides a shared memory abstraction over a group of physically distributed machines. It automatically handles data communication between machines and eliminates the need for the programmer to write message–passing code. A multithreaded Java program written for a single machine will require fewer changes to run on a Java/DSM combination than with RMI.

We have been implementing a JDK-1.0.2 based parallel Java Virtual Machine on the *TreadMarks* DSM system. Our implementation includes a distributed garbage collector and supports the Java API with very few changes. In this paper we will describe our motivation and the implementation, and report our early experience with programming under both RMI and DSM.

1 Introduction

Heterogeneity exists in many computing environments. It is therefore highly desirable to integrate heterogeneous machines into a coherent distributed system and to share resources among them. A number of methods for programming heterogeneous environments have been proposed in the past, including message passing libraries, Remote Procedure Call, and heterogeneous Distributed Shared Memory systems, but they failed to hide the hardware differences and/or the distributed nature of the system from the programmer. In this paper we propose to use the combination of Java and a Distributed Shared Memory (DSM) system as a platform for heterogeneous computing. This new platform transparently handles both the hardware differences and communication between physically distributed machines, making programming much easier than previous approaches.

DSM systems are motivated by the desire to simplify the programmer's task by handling some of the low-level program details automatically in the run-time system. A DSM system handles the communication of data between machines, eliminating the need for the programmer to write message-passing code. Roughly speaking, a DSM system enables processes on different machines to share virtual memory, even though no physical memory is shared by the machines [7]. It is widely accepted that it is easier to program with shared memory than message passing: Instead of sending and receiving messages explicitly, programs can use ordinary loads and stores to access shared data. This enables programmers to concentrate on algorithmic issues rather than on managing partitioned data sets and communicating values.

One of the most important features of Java is its portability. Java compilers generate code in an architecture–independent format so that the same code can be executed on many different machines, given the presence of the Java Virtual Machine. There are no implementation–dependent aspects of the Java language specification. Therefore an object will present the same view to the programmer regardless of the machine architecture and the implementation of the Java Virtual Machine. With Java, we can expect an application developed on one type of machines to run on another type of machines without modification, and exhibit exactly the same behavior. In short, for a large class of applications, Java will be the language of choice in heterogeneous environments.

Java provides the Remote Method Invocation (RMI) mechanism for distributed computing. RMI is an enhanced Remote Procedure Call (RPC) mechanism that provides additional support for the use of object references across machines. When an object is passed across machines as an argument to or the result from a remote procedure call, either call–by–reference or call–by–copy semantics can be used. The choice is made by the programmer on a per class basis at compile time. Our experience with RMI has shown, however, that this additional support for linked structures does not eliminate the need for reference marshaling in all cases. Besides, programming with an RPC mechanism also requires significant effort from the programmer in other areas, namely the coherency management of replicated data, and the fine tuning of the remote interface to improve communication efficiency. These efforts are not trivial even for experienced programmers. With the coming of Java we have seen people who have never written a distributed application starting to do so. We believe it is highly desirable to provide them with a simpler programming platform.

We are implementing a JDK-1.0.2 based parallel Java Virtual Machine on the TreadMarks DSM system [6]. Our implementation includes a distributed garbage collector and supports the Java API with very few changes. To program our our implementation is very much like writing a multi–threaded Java program on a single processor. Our early experience with the prototype showed that the combination of Java and DSM can hide both the hardware differences and the distributed reality of the system from the programmer, providing greater transparency than existing systems.

The rest of this paper is organized as follows. Section 2 compares our work with existing approaches for heterogeneous computing. Section 3 describes our design in detail, and summarizes the changes made to the Java Virtual Machine. In Section 4 we report the implementation status and discuss our programming experience with both RMI and the Java/DSM combination. Our concluding remarks are offered in Section 5.

2 Motivation

A number of systems have been proposed for heterogeneous computing environments. They can be divided into two categories: those based on message passing and those which provide a shared memory abstraction. In this section we first discuss the problems that arise in a heterogeneous environment. Then we examine how the existing methods and the Java/DSM method deal with these problems.

2.1 Issues to Address

When designing a platform for heterogeneous computing, two basic issues need to be addressed:

- Hardware differences.
  In heterogeneous environments, data items may be represented differently on various types of hosts. For example, the size and byte ordering of integers may be different; and the lengths of the mantissa and exponent fields as well as their positions in a floating point number can differ. For each data item, the alignment requirements can differ too. There are two questions here:

  - Can the data types, both basic and aggregate, be automatically converted between machines?
  - For aggregate data types, will they have the same size and ordering of fields on all machines?

  If automatic conversion or a uniform view is not achieved, the programmer will have to identify the architecture–dependent part of the program and write different code for each architecture. This will be a tedious and error–prone process.

- Distributed nature.
  A heterogeneous system is usually a distributed system too, in which a group of machines are connected by a communication network. Programs running on different machines must send messages over the network to communicate with each other. Furthermore, since pointers are usually meaningless across machine boundaries, the sharing of complex data structures usually requires first marshaling them into a byte stream, then reconstructing them on the remote machine.

2.2 Heterogeneous DSM Systems

Several heterogeneous DSM systems have been designed in the past, including Mermaid [13] and Agora [3]. We take Mermaid as an example.

Mermaid is implemented on the IVY DSM system and supports the C language. Because of the shared memory abstraction Mermaid provides, the distributed nature is hidden from the programmer. The programmer can use ordinary loads and stores to access remote data; and exact knowledge of the data access patterns is not required to make the program work. Complex structures can be accessed using pointers.

The hardware differences, however, are exposed to the programmer. For each basic data item, the programmer sees the difference in its sizes on different machines. Aggregate structures are padded so that they are of the same size on all machines, but the padding and the possible difference in the ordering of fields are visible to the programmer. Mermaid can automatically perform data format conversion between machines. The only requirement for the programmer is to declare the type when allocating an object. But the automatic conversion requires non–standard compiler support.

2.3 Message Passing

Message passing has been the prevailing programming paradigm in distributed environments both homogeneous and heterogeneous. Message passing based platforms include message passing libraries and Remote Procedure Call (RPC).

2.3.1 Message Passing Libraries

Message passing libraries such as PVM [4], TCGMSG [5] and Express [9] have long been in use. A message passing standard MPI [8] has also been developed.

PVM provides basic primitives such as *send*() and *receive*() as well as routines to convert basic data items between native formats and a standard format. Before a basic data item is sent to a remote machine, the programmer must call a suitable *pack*() routine to convert it to the standard format. For aggregate types, this call must be made for every field in it. On the remote machine, corresponding *unpack*() calls are made to retrieve data from the message.

PVM and other message passing libraries do not support the sharing of complex data structures such as linked lists. It is up to the programmer to marshal them into a message and reconstruct the data structure on the remote machine.

2.3.2 Remote Procedure Call

In the Remote Procedure Call (RPC) [2] model, servers export interfaces, making a specific set of operations available to clients that bind to the corresponding interface. Clients then invoke these server operations by making RPCs. To the client, the RPC is supposed to look like a normal procedure call. In reality, the client actually invokes a stub that marshals the arguments into a message, delivers the message to the server, waits for a reply message, unmarshals the result from the reply, and returns it to the client. Such systems typically define a network standard data format for the call and reply messages and all machines convert data between their local representations and this standard format.

The RPC model simplifies the sending and receiving of messages. Data items are automatically marshaled into messages at the sender, and unmarshaled from messages at the receiver. Data format conversion is done automatically. But the programmer must define the RPC interface in detail, including the type of every argument for a particular call. The difference between representations of aggregate types is visible too.

With the RPC model, like with message passing libraries, the programmer must have full knowledge of data access patterns and decide *when* to communicate with other machines, *whom* to communicate with, and *what* to communicate. When the data access pattern is unpredictable, the programmer must be conservative and may send more messages than necessary.

The Java RMI mechanism supports the use of object references across machines. The parameters to and the result returned by a remote procedure call may be passed by reference or by copy. In the former case, a proxy object is created on the remote machine, and a reference to the proxy object is passed to the program running on that machine. In the latter case, the entire object graph rooted at the named object will be automatically converted into a byte stream on the host and reconstructed on the remote machine. The programmer decides which semantics to use on a per class basis at compile time.

Although the support for cross–machine object references allows sharing of linked structures in a distributed environment, in practice it is often inefficient. For example, when a program wants to traverse a linked list on a remote machine, the access to every entry in the list will invoke a remote procedure call. It would be more efficient if the entire list is fetched at once. This is the reason why the call–by–copy semantics is provided. However, this approach creates disparity between local and remote calls, since local calls use only call–by–reference. Furthermore, the call–by–copy implementation may create new inconvenience to the programmer, as we will discuss in Section 4.4.3.

2.4 Java/DSM

Unlike previous heterogeneous DSM systems, the Java/DSM combination completely hides the hardware differences from the programmer. Because there are no implementation–dependent parts of the Java language specification, an object presents the same view to the programmer regardless the underlying machine architecture. Data conversion is done automatically without any effort from the programmer. There is no need for non–standard compiler support.

Compared with the message passing based platforms, the combination of Java/DSM enjoys the benefit of a shared memory abstraction: transparent handling of message passing and easy sharing of complex structures.

The use of Java also provides another benefit. Because Java programs are compiled into an architecture–independent format, the same code can be executed on any architectures.

2.5 Summary

There are two basic issues to address in a heterogeneous environment, the hardware differences and the distributed nature of the system. Failure to hide hardware differences forces the programmer to identify all architecture–dependent parts of the program, a very tedious and error–prone process. When the distributed nature is exposed to the programmer, the programmer must know more details about the data access pattern. Complex data structures are hard to handle.

Several methods have been proposed for heterogeneous computing, including heterogeneous DSM systems, message passing libraries, and RPC. None of them hides the hardware differences from the programmer, and the latter two also expose the distributed nature of the system. In contrast, the combination of Java and DSM can handle both issues transparently.

3 Design

Our design is based on the JDK-1.0.2 Java Virtual Machine. In this section we discuss three issues: user interface, memory management, and data format conversion. We will also summarize the changes made to the Java VM.

3.1 User Interface

In our implementation, there is one Java Virtual Machine on each machine. This Java VM is very similar to the JDK Java VM, except that all objects are allocated in the shared memory region. Our implementation supports the Java API. In our current implementation, a thread's location is not transparent to the programmer, and threads cannot migrate between machines.

In RMI, a remote object can only be accessed through methods defined in a remote interface. In our implementation, however, a remote object can be used in the same way as a local object, including the invocation of every method defined in the class and the load and store of static and instance variables. In particular, the static variables will be shared by all objects in the system.

3.2 Memory Management

In a DSM system, an object created on one machine may be referenced by a process on another machine. Since communication is relatively expensive in distributed environments, it is impractical to trace the entire object graph spanning over the system. It is highly desirable that each machine can perform garbage collections independently, yet a machine must know all remote references so that objects only referenced by remote machines will not be prematurely reclaimed.

Here we give a brief introduction to our garbage collection algorithm. A detailed description can be found in [12]. The garbage collector on each machine maintains two lists: the *export* list, which contains remote references to locally created objects; and the *import* list, which contains references to remote objects. These two lists are not accurate. They are a conservative estimate of the real cross–machine reference set.

Before a message is sent to another machine, the DSM runtime makes an upcall to the garbage collector and passes it the contents of the message. The garbage collector checks the message for valid references to local objects and inserts them into the export list. When a message arrives, the DSM also makes an upcall and the garbage collector looks in the message for valid remote references to put in the import list.

During each garbage collection, the export list is treated as part of the root set of references, and there is no need to communicate with remote machines. Imported references that are not marked will be sent back to their owners. The owner uses *weighted reference counting* [10, 11, 1] to decide when a reference can be removed from the export list.

This algorithm allows each machine to perform garbage collections independently most of the time, although sometimes synchronized collections are required to reclaim cyclic structures.

3.3 Data Conversion

In heterogeneous environments, data items may be represented differently on various types of machines. When a data item is passed between machines, data conversion is required. How the conversion should be done depends on the type of the data. For example, between a big–endian and a small–endian machine, a 4-byte string needs no conversion, but an integer should have its bytes reversed.

3.3.1 Type Lookup

To perform data conversion on an arbitrary address range, we must find out the types of the data involved. In the JDK-1.0.2 virtual machine, an object consists of two parts, the handle and the body. The object body holds the real data. The handle has two fields, a pointer to the body, and a pointer to a structure which contains the type of every field in this object. Therefore, given a data item, if we can find the handle for the enclosing object, we can determine its type.

To efficiently identify the object to which an arbitrary data item belongs, we require that objects allocated from the same page be the same size. We also add an extra field to the object body: A back pointer to the handle. By following the back pointer we can locate the type descriptor of the object. Thus, given any address, we can identify the page number, look up the size of the objects contained within this page, find the beginning of the object the address belongs to, and follow the back pointer to find the type information.

A special case is when a data item crosses a page boundary. The allocator ensures that objects are long–word aligned so that no basic data types will cross a page boundary. Complex objects may, however, cross a page boundary. To handle this case, we record for each page how many bytes in the front of the page are used by an object crossing from the previous page. We will need to subtract this value when calculating the starting address of an object.

The following code fragment illustrates how to convert the data items in a page. For simplicity's sake, assume the first object starts on the page boundary.

```
page_number = (page_start - first_shared_page) / page_size;
object_size = page_info[page_number].size; /* size of objects on this page. */
```

```
addr = page_start;
while (addr < next_page;) {
   back_pointer_address = addr + back_pointer_offset;
   back_pointer = (JHandle*) (*(long*) back_pointer_address);
   class = back_pointer->class;

   convert_pointer(back_pointer_address);
   addr += sizeof(long*);   /* Locate the start of data. */
   for (i = 0; i < class->fields_count; i++) {
      type_code = class->fields[i].signature;
      convert(addr, type_code);
      addr += field_length(type_code);
   }
}
```

As a side note, the pages we have been referring to are logical pages that are of the same size on all machines. The difference in hardware page sizes is handled transparently by TreadMarks.

3.3.2 Missing Type Information

TreadMarks is a page–based DSM system, that is, when an object is accessed by a remote machine, all data on the same page will be sent. For an object that spans two or more pages, it is possible that the rear end of the object arrives at a remote machine before the front end, which contains the back pointer to the type information. In Java VM implementations like JDK-1.0.2's, where the handle and the object body are separate, we may also find the handle missing.

When the type information is missing, we cannot simply retrieve the missing page, for two reasons:

- The missing page may never be used by the application. We want to avoid getting unnecessary data since communication cost can be high in distributed environments.
- In the case of page–crossing objects, if we retrieve the previous page, it may have an object crossing from its preceding page too. We may end up retrieving a series of pages.

When the back pointer of an page–crossing object is missing, we do not convert the part of it we received. We store it as is and record from which machine it came. If the object is used on the remote machine, its handle must arrive before the first usage. When we see a handle to a page–crossing object, we verify if the object has been properly converted. If not, we can do it then. This work can be performed in the upcall to the garbage collector discussed in 3.2, which is invoked on every incoming data message.

3.4 Changes to JVM

We have made the following changes to the JVM:

1. The heap is allocated using TreadMarks' shared memory allocation routine.
2. Classes loaded by the JVM are put in shared memory. The loading is synchronized.

3. During the garbage collection, the garbage collector must recognize objects created by remote machines, mark those referenced locally, and send the unused remote references to their owners after the collection.

4. To support data conversion, we require a mapping from an arbitrary address to the object's handle. The straightforward solution is to require that all objects on a page are of the same size, and put a pointer to the handle at the beginning of every object.

From the list we can see that most of the changes are made to the memory management module. The rest of the Java VM is virtually unchanged.

4 Early Experience

In this section we will report the current status of our implementation and discuss our experience with programming with both DSM and RMI.

4.1 Current Status

We have modified the garbage collector to allow each machine to perform independent garbage collections. The synchronization primitives have not been fully implemented. TreadMarks' *lock*() will be invoked automatically for methods with the *synchronized* specifier, but the two synchronization primitives, $wait()$ and $notify()$, do not work between threads running on different machines.

Applications that do not use $wait()$ and $notify()$ to synchronize can already run on a group of Sparc workstations. Our next step is to make it run in a heterogeneous environment consisting of Sparc workstations with Solaris and Pentium processors with FreeBSD.

4.2 Sample Application

We are developing a distributed spreadsheet based on the public–domain spreadsheet calculator *sc*. The functionality of the distributed spreadsheet has been simplified, but the internal data structures are very similar to *sc*.

The distributed spreadsheet provides the basic functionality of *sc* with additional extensions supporting collaboration between multiple users. The extensions include:

- Define regions of cells.
- Assign read/write privileges for each region to users.
- Lock a region to prevent others from writing it.
- Check if other users have made changes.
- Incorporate changes made by other users.

The internal data structures of our spreadsheet are similar to that of *sc*. The cells are organized in a two–dimensional array. Each cell entry contains information such as its type, current value, printable form, and a lock bit. If the content of a cell is an expression, an E–tree is created from the expression for the cell. An E–tree is a binary tree whose leaf nodes are constants or references to cells, and whose intermediate nodes are operators.

4.3 DSM Version

The DSM version is very easy to program. All we needed to do was to identify race conditions and add synchronization. For example, the accesses to cells and the checking/assignment of access permissions must be synchronized.

4.4 RMI Version

4.4.1 Data Replication

In the RMI version, the first decision we must make is on data replication.

Without data replication, the program would be much simpler because the programmer would not have to worry about data coherency. However, there are two problems. First, a centralized server does not scale well with the number of machines in the system. Second, the programmer might have to fine–tune the RPC interface to improve communication efficiency. In our first experiment, every cell in the spreadsheet was exported as a remote object. When a client machine wanted to refresh its window, it had to make a separate remote call to get the printable form for every cell. With a 10x10 window, the user could observe a significant slowdown. To improve the performance, we added a new call to the RPC interface that would return the printable forms of all cells in a given rectangle.

With data replication, we have to maintain the data coherence between the copies on different machines. There are two schemes to choose from: The *update* scheme, in which a machine making changes to the spreadsheet actively updates others; and the *invalidate* scheme, in which a short invalidation message is sent instead of all the changed data.

The *invalidate* scheme invites many problems such as maintaining different versions of the datasheet, keeping track of what versions have been sent a remote machine, and how to incorporate changes from two or more remote machines. A complicated timestamp mechanism may be needed to keep track of the ordering of events.

The *update* scheme is much simpler, but it may result in redundant communication. Since one machine cannot anticipate the user activity on another machine, it has to push updates to remote machines regardless of their need.

Our final version uses data replication and the *update* scheme.

4.4.2 Reference Marshaling

Although the RMI package allows for both call–by–reference and call–by–copy semantics, with the use of replication, there are still cases where reference marshaling is unavoidable. The example in Figure 1 illustrates the problem. On machine 1, the cell A[0] contains an expression. When we want to propagate A[0] to machine 2, we need a E–tree that is identical to the tree on machine 1, except that the leaf nodes should point to the cells on machine 2, like the dotted lines do in Figure 1. However, whether we choose pass–by–reference or pass–by–copy semantics for the cells, the leaf nodes of the E–tree on machine 2 will not point to the desired places, as shown in Figures 2 and 3. The programmer will have to traverse the tree and adjust them.

4.4.3 Bundling of Linked Objects

RMI's call–by–copy semantics provides useful support for linked structures. In Figure 1, when the expression–tree is passed between two machines, it makes sense to group the nodes in the tree and send them together. However, in some cases it may introduce new problems for the programmer. Assume we have a linked list, and the objects in the list use the call–by–copy semantics. With

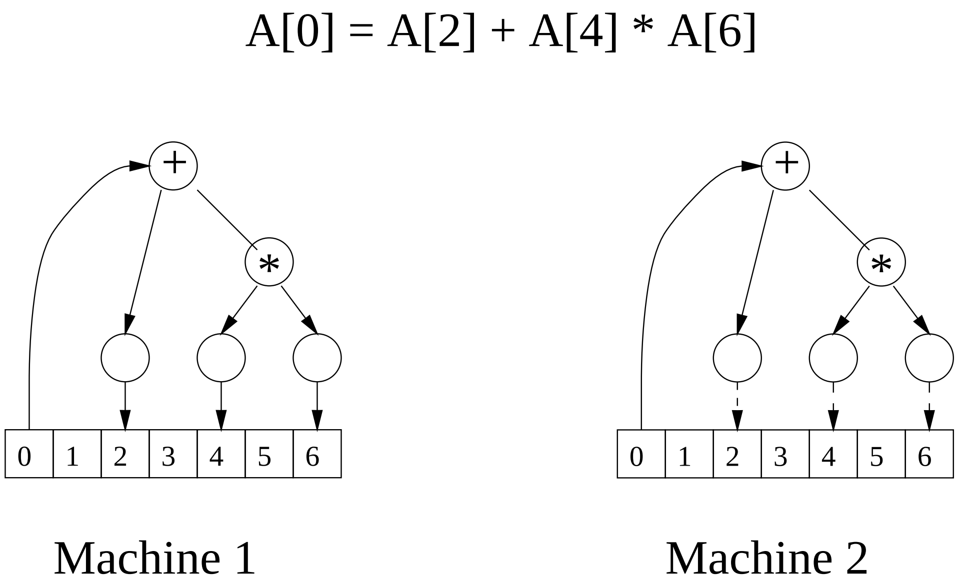

Figure 1 Reference Marshaling.

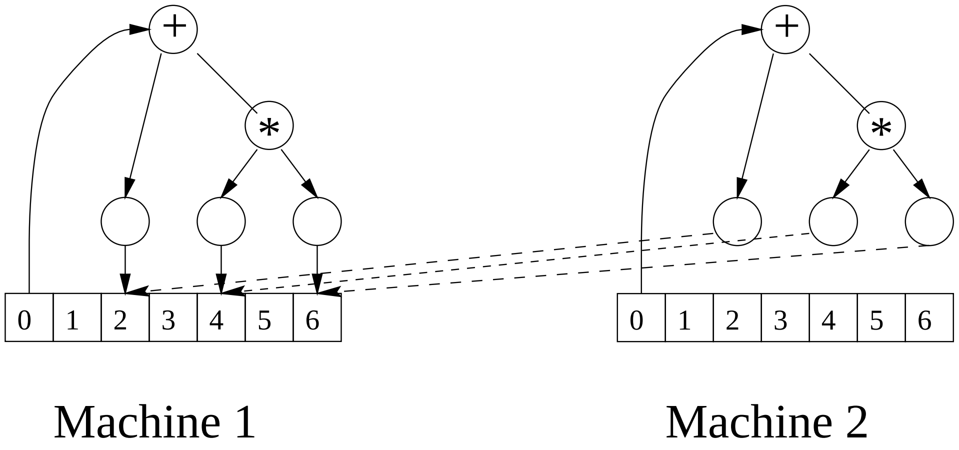

Figure 2 Using Pass–by–Reference.

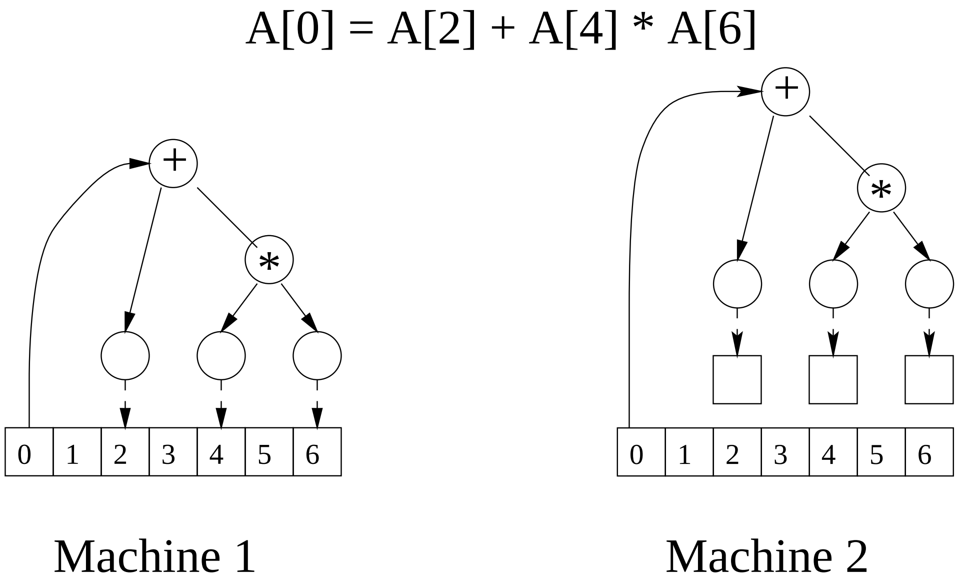

Figure 3 Using Pass–by–Copy.

RMI, if an object is passed to a remote method, all objects behind it in the list will be passed too. In the extreme case, if the head of the list is passed as an argument to the remote call, the entire list will be sent. If communication cost is a concern, the programmer may have to make a separate copy of the entry, then pass the new copy as an argument.

This example, and the reference marshaling problem mentioned in Section 4.4.2 illustrate the difficulty of providing efficient support for linked structures in an RPC–based system.

4.5 Summary

In this section we described our experience with developing a distributed program using Java's RMI package and using the TreadMarks DSM system. With the RMI package, the programmer is exposed to the distributed nature of the system and a lot of extra work may be necessary, namely maintaining the coherency of replicated data, optimization of the remote call interface, and reference marshaling. With the shared memory platform, all of these issues are implicitly handled by the DSM system and the program is much easier to write.

5 Concluding Remarks

In this paper we presented our ongoing work to implement a parallel Java Virtual Machine that runs on the TreadMarks DSM system. Compared with existing approaches for heterogeneous computing, this new platform provides greater transparency to the programmer: the use of Java hides the hardware differences from the programmer; the DSM system transparently handles the message passing details such as data replication, remote interface design, and reference marshaling. In particular, our experience with RMI showed that it is hard for RPC based models to provide efficient support for the use of linked structures, while the DSM system can handle it easily. We believe the combination of Java and DSM provides a good platform for heterogeneous computing.

References

[1] D. I. Bevan. Distributed garbage collection using reference counting. In *Parallel Arch. and Lang. Europe*, pages 117–187, Eindhoven, The Netherlands, June 1987. Spring-Verlag Lecture Notes in Computer Science 259.

[2] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[3] R. Bisiani and A. Forin. Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8), August 1988.

[4] G.A. Geist and V.S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, pages 293–311, June 1992.

[5] R.J. Harrison. Portable tools and applications for parallel computers. In *International Journal of Quantum Chemistry*, volume 40, pages 847–863, February 1990.

[6] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

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

[8] Message Passing Interface Forum. MPI: A message-passing interface standard, version 1.0, May 1994.

[9] Parasoft Corporation, Pasadena, CA. Express user's guide, version 3.2.5, 1992.

[10] R. Thomas. A dataflow computer with improved asymptotic performance. Technical Report TR-265, MIT Laboratory for Computer Science, 1981.

[11] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, Eindhoven (the Netherlands), June 1987. Springer-Verlag.

[12] Weimin Yu and Alan Cox. Conservative garbage collection on distributed shared memory systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[13] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5), September 1992.