Java versus MPI in a Distributed Environment

Maurice Eggen
Computer Science
Trinity University
San Antonio, Texas 78212

Roger Eggen
Computer and Information Sciences
University of North Florida
Jacksonville, Florida 32224

Abstract *Networked unix workstations as well as workstations running Windows 95 or Windows NT are fast becoming the standard computing environments at many universities and research sites. Researchers and educators seek simple methods to harness the potential for parallelism implicit in these computing networks. This paper investigates the ease and efficiency of Java sockets, Java remote method invocation (RMI), and Message Passing Interface (MPI) to address these needs. We look at the ease of programming in each of the environments and efficiency considerations using sequential programs as a base mark. For certain applications, Java may not be the best choice, but in other cases, Java provides a simple, robust distributed programming environment.*

Keywords

Java, message passing interface, distributed and parallel programming

1 Introduction:

Networked unix workstations as well as workstations based on Windows 95 and Windows NT are fast becoming the standard for computing environments in many universities and research sites. In order to harness the tremendous computing capability represented by these networks of workstations, researchers and educators seek simple tools which will allow experiments in distributed and parallel computing.

Java, with all of its followers, is fast becoming a major player in this arena. With multiple tools, Java, since it is a simple language to learn, has become the choice of many programmers when developing distributed applications. This paper will discuss the efficiency and effectiveness of Java compared to the Message Passing Interface standard (MPI) for developing such applications.

Since personal workstations are becoming more powerful and network speeds have improved, it becomes more and more desirable to harness the potential a network of workstations presents to the educator or researcher. Moreover, since the cost of such workstations is fast approaching the price of sand, every university, no matter how small, can afford at least a small network of workstations, either running linux, or perhaps Windows NT. It is no longer necessary for universities to spend hundreds of thousands of dollars to purchase high-end parallel computing hardware in order for such experimentation to take place.

Several software programming environments are beginning to emerge as standards in distributed parallel processing (DPP). For a heterogeneous cluster of unix workstations, parallel virtual machine (PVM) enjoys wide usage. An attempt to standardize the message passing inherent in DPP, message passing interface (MPI) was developed. Both of these programming environments effectively harness the capability of a distributed network of unix workstations.

This paper will show how easily Java can be applied to DPP. We will show how common parallel applications may be studied using Java, and compare their implementation in Java to their implementation in MPI. We discuss Java's tools for DPP, and compare the efficiency and effectiveness of Java compared to MPI. We discuss the availability, hardware and software issues, and programmer interface.

2 Hardware

The hardware for this experiment consisted of five 233 MHz pentium machines running RedHat linux v5.1. They were connected by a 100 megabit fast ethernet. Additional machines were available, including Hewlett Packard, SGI, Macintosh and several Windows NT machines, but this project was limited to the five pentium machines in order to control the many variables involved.

3 Software

The idealized model for a parallel machine is that of a multicomputer. Several von Neumann machines are connected via some sort of interconnection network. Whether the von Newmann machines are tightly coupled or exist on some network, the model applies. The various nodes on the network communicate via some sort of messaging scheme. The only difference between the network model and the tightly coupled model is the cost of sending a message between the nodes.

Since messaging is the focus for the ability of the multicomputer to be able to access (read and write) remote memory, various schemes for accomplishing messaging have been developed. Because we believe messaging is important to any discussion of parallel and distributed processing, we focus on the capabilities of messaging provided by both Java and MPI. Of particular importance is the fact that these software environments are available in the public domain and can be used on a wide variety of platforms.

3.1 Needs of a Messaging System

Messaging systems are subject to certain requirements in order to provide the capability to perform general purpose concurrent execution of processes:

- ♦ A process is itself inherently sequential code. Processes must have the ability to communicate with a larger environment; that is, read and write remote memory.
- ♦ A process must be able to send messages and receive messages.
- ♦ A send operation should be synchronous. Processing should continue once a message is sent.
- ♦ A receive operation should have to ability to be asynchronous. It should be able to block the execution of the process until a desired message is received.
- ♦ A process should have the ability to perform dynamic task creation and allocation. Messaging should be able to be created and destroyed dynamically.

3.2 Message Passing

When a process, say process one, sends a message to a process, process two, whether under explicit control of the programmer or not, the following must happen:

- ♦ Process one must prepare a send buffer
- ♦ Process one must pack the relevant information into the send buffer
- ♦ Process one must initiate a send
- ♦ Process one sends the buffer
- ♦ Process two receives the buffer
- ♦ Process two unpacks the information from the buffer
- ♦ Process two performs the appropriate computations on the information
- ♦ Process two prepares a send buffer
- ♦ Process two packs the new information into the send buffer

- ♦ Process two sends the information to process three etc.

Whether this process is handled explicitly by the programmer or implicitly by the programming environment is in some sense irrelevant. The point is that it must be done for messaging to occur.

4 Java Availability and Installation

Several excellent Java developments are available commercially. The scope of this paper does not allow us detailed discussion of these products. For our work we chose the public domain version of Java available via download from Sun at http://www.sun.com/java. We found that its installation on our network of linux workstations was straightforward.

5 MPI Availability and Installation

Also available via download, a public domain version of MPI can be obtained from http://www.mcs.anl.gov/mpi. This version, called MPICH, is easy to install in most unix systems and requires only basic unix knowledge. MPI is a relatively mature software environment, and, with recent releases of the software, enjoys a rather wide usage and trouble free execution.

6 Comparison

To illustrate the features of the Java programming language in the DPP environment as well as compare the capabilities of Java to other messaging systems, a simple problem was posed, that of parallel sorting. We used the five machines described above as a distributed parallel machine and executed a simple parallel sort. The sort was implemented and timed in four different environments: MPI; Java parallel using sockets; Java parallel using remote method invocation (RMI). We implemented the sort in native Java for comparison purposes.

6.1 Java Sockets

As we stated in section 3.2 of this paper, distributed communication requires the first process to create a buffer, and initiate a send. The receiving process must accept the buffer and unpack the data. Socket communication in Java differs slightly from this general scenario.

In Java, it is possible to "serialize" a data structure. This simply means that the data as well as the structure is packaged by the language to be stored on an external device, such as a hard drive. Several common data structures, including arrays and dynamic lists, can be serialized. In our application, we serialized an array which was written to a socket.

To do socket communication in any language the programmer must establish the socket number to communicate through. The server establishes the socket from which it will listen and the client, requesting services from the server then writes to that socket number.

Java, through its ObjectInputStream and readObject facilities can send and receive a serialized data structure, essentially packing and unpacking or "deserialize" the data communicated.

Writing to a socket is much like writing to a file. Normally, when communicating through a socket a programmer must send small units of data, but with serialization Java is able to send complete data structures. In our experiment the client sent portions of an array to each server which sorted and returned the array. The server sends the array back to the client with the following instructions:

```
ObjectOutputStream out =
new ObjectOutputStream
(incoming.getOutputStream());
      out.writeObject(a);
```

where `incoming` was bound to the socket.

The communication via sockets is easy and efficient, but the programmer is responsible for packaging, sending, receiving, unpacking each data structure included in a message.

6.2 Java Remote Method Invocation

Unlike socket communication discussed in 6.1, the Java remote method invocation (RMI) is a higher level form of communication. RMI is built upon sockets, but the programmer is not concerned with establishing the socket through which communication occurs. The RMI establishes method invocation in much the same way Java methods are normally invoked. The difference is that the method to be invoked resides on a remote machine, the server.

Our experiment uses several servers, each running the same program. The client distributes portions of an array to each server which sorts and returns the sorted portion to the client. The client merely puts the array back together creating a complete sorted array.

In our experiment, we view the client as establishing an array of integers passed to several servers which sort, just we did in the socket program. Only now, we simply invoke the sorting method existing on the remote machine and pass the array as an argument. However, the equivalent of serializing the array occurs in this form of communication through marshaling the data via stubs and skeletons. The stubs and skeletons act as local versions of the remote method during program translation thus allowing the class files to exist even though the actual method is not present. The stubs and skeletons are created through execution of the rmic program provided with the Java Development Kit.

The rmiregistry must be running on the server. This program identifies to clients the remote method that resides on the server. The server provides the instruction that establishes the entry of the remote method in the rmiregistry through a Naming.rebind ("classname",s); instruction. This allows the client to find and invoke the remote method.

To perform client-server programming using RMI, the following, at a minimum, must be present:

- A server main program which binds the remote method to the registry.
- An interface defining the remote method’s parameters.
- An implementation which contains the remote method invoked by the client. The implementation is also used by rmic to create the stub and skeleton.
- A client program which determines the servers it chooses to communicate with, does a lookup in the registry to find the remote method, and invokes the method.

While there are more files that need to be created to take advantage of RMI after communication is established, remote methods are invoked in exactly the same manner as if they were local. The client-server aspect of programming becomes transparent at this point. Java provides easy access to the parallel applications through the Java.rmi.server.*, Java.net.*, Java.io.* classes.

6.3 MPI

Development of the Message Passing Interface (MPI) began in an effort to define a standard and portable message passing system that would support parallel applications and standards in a wide range of applications and environments. Currently MPI exists both as a commercial product and in the public domain. These products run both on tightly coupled, massively parallel machines and on networks of workstations. The applications in this paper were run on networks of workstations using MPICH, a public domain version of MPI.

MPI implements the single program multiple data (SPMD) paradigm of parallel programming. It is the responsibility of the programmer, using program logic, to allow processes to perform different tasks.

In our example, process 0 acquires an array, divides the array into five (nearly) equal parts, packs the parts into a send buffer, and sends a part of the array to each of the other processes participating. An MPI statement used to accomplish a send is given below.

```
MPI_Send(&a, length, MPI_INT, i, 0,
MPI_COMM_WORLD);
```

The first argument is the address of the array being transmitted, the second argument is the number of elements, the third is the data type of the elements in the array, the fourth is the process number of the process which is to receive the data, the fifth argument is a tag, and the sixth is a communicator which defines the processes running at the time the communication is initiated. The MPI_Send(), then, is responsible for preparing the send envelope, packing the data into the envelope, and sending the message to the receiving process.

It is then the responsibility of the remaining processes to receive the message, sort the portion of the array they are given, and return the array to process 0, which is acting as the host for this problem. A sample receive statement is given below:

```
MPI_Recv(&a, length, MPI_INT, i, 0,
MPI_COMM_WORLD, &status);
```

Each of the parameters is similar to the corresponding parameter in the MPI_Send(), with the exception of the status parameter, which can be used to determine information about the data that was actually received. The MPI_Recv() is then responsible for collecting the send envelope, retrieving the information to be processed, and unpacking the data into the appropriate variable locations. Thus, MPI_Send() and MPI_Recv() apply the principles discussed in sections 3.1 and 3.2 of this paper.

MPI_Send() and MPI_Recv() are relatively straightforward to use, being among the simpler of the commands MPI offers for message passing.

7 Results

The sorting experiment was run on the network of workstations described in section 2. Each of the timings was taken at 3:00am so that minimal network traffic was involved. Each of the timings is an average of five readings to minimize the network fluctuations. The chart below summarizes the findings. All times are in milliseconds.

Array Size	**Java Sockets**	**Java RMI**	**Java Sequential**	**MPI**
5000	636	5128	1841	119
10000	1694	10329	6864	333
15000	3343	18374	14648	602
20000	5525	29381	25183	998
25000	8252	42482	38336	1449
50000	30168	149866	144336	5072
100000	114602	554477	555069	18776

8 Summary and Conclusions

It is fairly clear, upon studying these data, that Java is not fast. Why, then, does one wish to program in Java for problems of this nature? We believe the answer is, firstly, Java is platform independent, and secondly, Java is extremely easy to use. With its extensive collection of methods for virtually any task that one wishes to perform, Java provides the programmer with a nice simple consistent programming environment independent of the hardware involved.

However, if speed is of paramount importance in a particular project, and if the project can be done in another language or programming environment, then perhaps Java is not the appropriate choice. Programming in MPI is easy as well. MPI, however, is not platform independent, and doesn't contain all the tools of the Java API. It is therefore up to the user to decide, based on the application, whether or not to choose one of these programming environments or the other.

9 Bibliography

[1] Harold, Elliotte R., Java Network Programming, O'Reilly Publishers, 1997

[2] Farley, Jim, Java Distributed Computing, O'Reilly Publishers, 1998

[3] Pacheco, Peter C., Parallel Programming with MPI, Morgan Kaufmann Publishers, 1997

[4] http://www.sun.com/java

[5] http://www.mcs.anl.gov/mpi