Abstract  Networked UNIX workstations as well as workstations running Windows 98 and Windows NT are fast becoming the standard computing environments at many universities and research sites. Additionally, many of these workstations, because of recent cost improvements, are commonly Symmetric Multiprocessor (SMP) workstations with two or four (or sometimes more) CPUs. Educators and researchers search for efficient methods of taking advantage of these advances in hardware technology. Java, with its wide usage, is fast becoming the choice of professionals because of its wealth of options as well as its ease of use. This paper investigates the use of Java sockets, Java RMI, and Java threads for parallel programming. We compare Java results with a standard message passing environment widely used in distributed programming, message passing interface (MPI).

Keywords  Java, message passing interface, distributed and parallel programming, sockets, threads, RMI.

1 Introduction

Networked UNIX workstations as well as workstations running Windows 98 and Windows NT are fast becoming the standard computing environments at many universities and research sites. Additionally, many of these workstations, because of recent cost improvements, are commonly Symmetric Multiprocessor (SMP) workstations with multiple CPUs. In order to harness the tremendous potential these powerful networks offer, researchers and educators search for effective and efficient software tools for programming these networks.

The Java development environment, because of its wealth of programming tools, has become the environment of choice for many applications. This paper will shed some light on some of the tools Java offers to take advantage not only of the SMP environment but also distributed clusters of workstations. We shall give comparative data illustrating the effectiveness and efficiency, as well as ease of use of Java, and will finally compare it to other standard ways of programming distributed clusters of workstations, most notably the message passing interface (MPI).

Since personal workstations and clusters of workstations are becoming more powerful and network speeds have improved, it becomes more and more desirable to harness the potential these workstations present to the educator and researcher. Moreover, since the cost of such workstations is fast approaching the price of sand, every university or research installation, no matter how small, can afford at least a small cluster 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 to study and take advantage of parallel and distributed processing.

One might argue that, with all of the advances in modern workstations, it is unnecessary to experiment with distributed and parallel processing technology. However, if advances are made in parallel and distributed programming techniques, and if workstation advances are made, these advances only serve to make the distributed cluster better.
This observation reinforces our desire for additional information concerning the programming of clusters of workstations and inexpensive SMP machines.

This paper will show how easily Java can be applied to distributed parallel processing, and how the Java threads programming tool can take advantage of SMP machines. Recent advances in operating systems (most notably recent releases of Linux) coupled with the Java development environment provide the programmer with efficient and effective methods of harnessing clusters of workstations as well as SMP machines. We shall compare these with a non-Java environment, C using MPI, a facility for message passing programming.

2 Hardware

The hardware for these experiments consisted several homogeneous workstations running RedHat Linux v6.0. The machines were all Pentium based PC’s including the two and four processor SMP machines. The machines were all connected by 100 megabit fast ethernet. The machines were 500 mHz pentium III processor based PC’s.

3 Software: Requirements

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 read and write remote memory.
- A process must be able to send messages and receive messages.
- A send operation should be synchronous.
- A receive operation should be asynchronous.
- A process should have the ability to perform dynamic task creation and allocation.

Messaging should be able to be created and destroyed dynamically.

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

- Process one must prepare a send buffer
- Process one must pack the 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 received 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 buffer 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.

3.1 Software: Java

The software for the Java portion of the experiment consisted of a standard linux based Java development environment. Widely available and easy to use, this environment is easy to install on UNIX (linux) machines. Available via download from Sun at http://www.sun.com/Java, we found the installation on our machines straightforward. Java is similarly easy to install and use on Microsoft operating systems.

3.2 Software: MPI

The version of MPI used in this comparison is MPICH. Available via download, this public domain version of MPI is easy to
install in most UNIX systems and requires only basic UNIX knowledge. MPI is a mature programming environment containing over 120 functions to control parallelism. However, most applications can be programmed using only six of the available MPI functions. Thus, it is easy to learn, robust, and yet comprehensive. It is trouble free and enjoys a rather wide usage.

4 Comparison

To illustrate the features of the Java programming language in the parallel environment as well as compare the capability of Java to other messaging systems, a simple problem was posed, that of parallel integer sorting. The cluster of workstations was envisioned as a distributed parallel machine, and was compared to two and a four processor SMP machines. Various problem sizes were considered and the sort was implemented in the following ways: Java distributed parallel using client-server socket communication for messaging, Java distributed parallel using remote method invocation (RMI), and finally MPI for message passing in the workstation cluster. For comparison purposes, Java’s threading libraries were used to take advantage of the two-processor and the four-processor SMP machines.

4.1 Java Sockets

In Java it is possible to "serialize" a data structure. This simply means that the data as well as its structure is packaged by the language to be stored on an external device, such as a disk drive. Several common data structures, including arrays and dynamic lists, can be serialized. In our application we serialize 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 stored and returned the array. The server sends the array back to the client with the following instructions,

```java
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, and unpacking each data structure included in a message.

4.2 Java RMI

Unlike socket communication discussed in the previous section, the Java remote method invocation (RMI) is a higher form of communication. The messaging is implicitly handled by the programming environment rather than explicitly by the programmer. It is the programmer’s responsibility to provide the appropriate environment for remote methods to be invoked, but once the environment is provided, the explicit messaging is handled by the language environment.

RMI is built upon socket communication, 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, which we may think of as 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 as 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 (packing) 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.*, and Java.io.* classes.

4.3 Java Threads

A thread is a single sequential flow of control. Thread objects allow multithreaded Java programming where a single Java virtual machine can execute many threads in a concurrent manner. A thread independently executes Java code that operates on Java values and objects residing in shared memory.

The single processor Java virtual machine must share its time between executing the code running in different threads. One such possibility for sharing is to give each thread a timeslice to use the processor. At the end of the timeslice, that thread must be suspended and the code for the next thread that is ready to run is given a slice of time.

If one has a multiple processor machine, the operating system must be capable of executing multiple threads concurrently. Using these machines, Java threads provide a natural and straightforward programming methodology to take advantage of parallelism and the available multiple processors.

A thread has a life cycle:

• A thread must be created (born).
A thread must be started. Depending on the availability of the processor(s) a thread may wait in a ready to run state.

A thread may be running (may actually be on one of the available processors).

After a thread has been running, it may go into one of several states:
- ready - will run when a processor is available
- sleeping - does not require a processor until it wakes up after time period
- suspended - does not require a processor until it is resumed
- waiting - not requiring a processor until notified that it may be transferred to a ready state
- blocked - not requiring the processor until some other event finishes

A thread may die when it is no longer required.

For our experiment, we implemented a threaded sort on two-processor and four-processor SMP machines. Every Java program contains at least one thread, the `main()` method. For the two processor machine, an additional thread was created; for the four processor machine three additional processes were created.

5 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. Constructed as a collection of libraries, MPI comes in both a C version as well as a Fortran version.

MPI implements the single program multiple data (SPMD) paradigm of parallel programming it is the responsibility of the programmer in an SPMD environment, 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:

```c
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 argument is the process number of the process which is to receive the data, the fifth is a tag, and the sixth is a communicator which defines the processes running at the time the communication is initiated. The `MPI_Send()` function is responsible for preparing the send envelope, packing the data into the envelope, and sending the message to the receiving process. `MPI_Recv()` is similar in nature with comparable arguments.

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

6 Results

The problem was run on the network and on the SMP machines described earlier in this paper. Each of the timings was taken at 3:00am when network influences would be minimum. Each of the times used in the computation is an average of five separate timings. All times are in milliseconds.
The "Java sequential" column represents a sequential Java implementation, whereas the "two threads" column represents a Java implementation using threads on a two processor SMP machine. The primary thread starts one additional thread, which processes half the data in parallel with the main thread, which processes the other half of the data. The "four threads" column is a Java implementation using threads on a four processor SMP machine. In this case, the main thread starts three additional threads, each of which process one-fourth of the data. The "client-server" column represents times obtained by establishing a socket connection to one server workstation, which processes half the data in parallel with the client workstation. The column entitled "client 3 servers" represents times obtained by establishing three socket connections to three server workstations, each of which process one-fourth of the data in parallel with the client (master) workstation. The column entitled "Java RMI, 3 servers" gives times obtained by establishing three remote servers, each of which is accessed in parallel from a client machine, which remotely invokes methods which assist in parallel with the solution to the problem. The last column entitled "MPI master-three slaves" is comparable, except that the message passing language used in the implementation was MPI rather than Java. The master passes one-fourth of the data to each of three slave workstations, each of which, along with the master, process one-fourth of the data in parallel.

### 7 Summary

Java RMI has considerable overhead associated with it. Its primary advantage is that it takes many of the nitty-gritty details out of the hands of the programmer. Since these tasks are essentially the same in all applications there is no reason for the programming environment not to do these tasks for the programmer.

Java sockets offer considerable improvement in efficiency over RMI. The disadvantage is that the programmer is now responsible for more of the tasks associated with the messaging. However, socket programming is not difficult and offers a relatively clean message passing programming environment.

If an SMP machine is available, the authors highly recommend the Java threaded environment in which to perform comparable programming tasks. Threads are easy to program, with a robust collection of tools in the Java.io.* and Java.util.* classes, and offer excellent ease of use and efficiency.

The C based MPI programming environment is the fastest of the distributed programming environments. However, MPI has the disadvantages that (a) the programmer is responsible for many of the details of the messaging and (b) MPI is not as portable as Java. In fact, there is no reason that the machines in this experiment were all Linux machines. They could have equally well been NT machines, or any other machine running Java. The portability of Java is a definite advantage.

#### Table 1 Experimental Results

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Java sequential</th>
<th>Java-two threads</th>
<th>Java-four threads</th>
<th>Java client-server</th>
<th>3 servers</th>
<th>3 servers</th>
<th>Java RMI</th>
<th>MPI-master</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1500</td>
<td>5520</td>
<td>12620</td>
<td>668572</td>
<td>72550</td>
<td>24995</td>
<td>87816</td>
<td>66867</td>
</tr>
<tr>
<td>10000</td>
<td>12620</td>
<td>5327</td>
<td>2395</td>
<td>1283</td>
<td>12542</td>
<td>72550</td>
<td>72550</td>
<td>72550</td>
</tr>
<tr>
<td>15000</td>
<td>33621</td>
<td>9117</td>
<td>14176</td>
<td>958</td>
<td>17682</td>
<td>19193</td>
<td>19193</td>
<td>19193</td>
</tr>
<tr>
<td>20000</td>
<td>134015</td>
<td>14176</td>
<td>17811</td>
<td>95322</td>
<td>26428</td>
<td>26428</td>
<td>26428</td>
<td>26428</td>
</tr>
<tr>
<td>50000</td>
<td>668572</td>
<td>198033</td>
<td>53613</td>
<td>371458</td>
<td>369941</td>
<td>369941</td>
<td>369941</td>
<td>369941</td>
</tr>
<tr>
<td>100000</td>
<td>134015</td>
<td>198033</td>
<td>53613</td>
<td>371458</td>
<td>369941</td>
<td>369941</td>
<td>369941</td>
<td>369941</td>
</tr>
</tbody>
</table>
8 Bibliography


