A Comparison of the Capabilities of PVM, MPI and JAVA for Distributed Parallel Processing

Roger Eggen  
Computer Science  
University of North Florida  
Jacksonville, Florida 32224

Maurice Eggen  
Computer Science  
Trinity University  
San Antonio, Texas 78212

Abstract: 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 potential for computing capability represented by these networks of workstations many new (and not so new) tools are being developed. Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) have existed on Unix workstations for some time, and are maturing in their capability for handling Distributed Parallel Processing (DPP). Recently, however, JAVA, with all of its followers, has begun to make an impact in the DPP arena as well. This paper will explore each of these three vehicles for DPP considering capability, ease of use, and availability. We will explore the programmer interface as well as explore their utility in solving real world parallel processing applications. We will show that each has its advantages. The bottom line is that programming a distributed cluster of workstations is challenging, worthwhile and fun!

Keywords: parallel and distributed processing, PVM, MPI, Java

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 universities and research sites. Workstations are becoming more powerful, less expensive, and generally easier to use than ever before. Because of the attractiveness of these workstations, and because of the fact that they represent tremendous unharnessed potential for computing, many people are beginning to harness clusters of workstations to effectively simulate parallel computers.

Since workstations are becoming increasingly more powerful, and since advances in network technology have improved the speed of the network itself, it becomes more and more desirable to attempt to harness the potential a networked laboratory of workstations presents to the researcher.

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 as well as a heterogeneous network of unix and windows NT workstations.

Because it encompasses a complete programming environment, JAVA, with its racehorse development, is enjoying considerable usage. With its remote method invocation (RMI), JAVA has become a significant player in DPP.

How do these programming environments compare? How easy is it to program one versus the other? What kind of programmer interface exists? What kind of workstations are necessary? Which programming system appears to be the most capable? What are the advantages and disadvantages of each? In this paper, we shall discuss each of these programming systems in some detail. We shall then offer our opinions of the answers to the above questions. Of course, one person’s gem is another’s garbage. The reader must ultimately decide upon programming preferences based on experience alone.

2 Hardware

The hardware for these experiments is rich in its diversity. There should be something here for
everyone. A network of Unix workstations consisting of 15 HP 9000 - 712 60 mHz workstations running HP UX 9.5 formed the bulk of the Unix workstation environment. In addition, however, there were two HP 9000 - 712 80 mHz workstations. A 200 mHz SGI workstation as well as a 160 mHz SGI were also included. For diversity, several 133 and 166 mHz pentium machines running RedHat Linux 4.2 were also included in the cluster. In all, one could configure a parallel machine consisting of three different architectures, with 6 different processor speeds as well as either a RISC architecture (HPPA or SGI5) or a CISC architecture (pentium).

Windows NT workstations were also involved in the experiment. A windows NT server (200 mHz pentium) provided the networking backbone. Additionally, five NT machines (133 and 166 mHz pentium machines) were part of the network.

3 Software

The idealized model for a parallel machine is that of a multicomputer. Several von Neumann machines are connected via an 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 a 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 seems to be 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 devised. Supporting evidence for messaging is included in the following quote from a young professional in the field:

"If you have time every now and then, let me know how the Parallel Processing research goes. The TWC work I’ve been involved with at SwRI has required some Parallel Processing. We’re running 19 processes on a 4 processor Onyx. The load balancing and synchronization issues have been quite a task. We had a lot of problems with using shared memory, and the team decided to use a message passing scheme via sockets for inter-process communication. I wasn’t actually involved in that decision, but the group at the time thought it was the way to go. The work has been interesting."

Because this author believes that messaging is important to any discussion of parallel and distributed processing, we shall focus on the capabilities for messaging provided by some of the important software development tools available in the market today. 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 hardware platforms.

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 the 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.

We shall now discuss each of the chosen software tools and explain how each satisfies the requirements for messaging systems. We shall also discuss how to obtain the software, some
general comments concerning its installation, and comment on each packages’ ease of use.

4 Parallel Virtual Machine: Availability

PVM is in the public domain. As a relatively mature software environment for parallel processing, PVM enjoys an extensive collection of ancillaries and libraries which may be obtained from several sites. The PVM home page at http://www.epm.ornl.gov provides links to most of the sites. Downloading the source is straightforward.

Installation on most Unix platforms is relatively easy (students have performed the installation at our institutions). PVM enjoys wide applicability as well. It has been successfully been installed on many Unix platforms, including Silicon Graphics, Hewlett Packard, IBM, Linux, and many others, as well as Windows 95 and Windows NT. The installation on the Windows operating systems is not as straightforward, but can be accomplished by someone who knows the quirks of these operating systems. Thus, PVM enjoys the capability to perform experiments in parallel processing on virtually any heterogenous cluster of workstations as well as many shared memory parallel processing machines.[4]

5 Parallel Virtual Machine: Programming

Designed as a collection of C, C++ or Fortran functions, PVM is a comfortable environment for an experienced programmer. Moreover, after having an introductory class in programming, students find the environment accessible. The messaging requirements are provided by specialized functions which accomplish the buffer creation, the data preparation, the sending and the receiving of the data. For example, the following code fragment illustrates a process packing an array of integers of size Size, and sending that array to a receiving process (presented in C):

```c
pvm_initsend(PvmDataDefault);
pvm_pkint(&Size,1,1);
pvm_pkint(array,Size,1);
pvm_send(tid,msgtag);
```

Here, the pvm_initsend() function prepares a send buffer, while the pvm_pkint() packs an array into the buffer, and the pvm_send() function sends the buffer to a task with PVM task identifier tid. the tag msgtag is used to assist in identifying appropriate messages.

In the receiving process, the process with PVM task identifier tid, there must be a corresponding receive, which will wait until it hears from the sending process. The receiving code might appear as follows:

```c
pvm_recv(tid,msgtag);
pvm_upkint(&Size,1,1);
pvm_upkint(array,Size,1);
```

The pvm_recv() is blocking, so that it satisfies the requirement of being asynchronous. Thus, PVM processes can accomplish the requirements set forth above. It can dynamically create processes with the function pvm_spawn() and can communicate with its environment with the send and receive commands.

6 Message Passing Interface: Availability

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 also a relatively mature software environment, and, with recent releases of the software, enjoys a relatively bug-free execution. [1]

7 Message Passing Interface: Programming

Also designed as a collection of C and Fortran libraries, MPI, like PVM, is relatively easy to program. MPI attempts to standardize the message passing environment.

Like programming PVM, MPI also provides functions MPI_Send() and MPI_Recv() to facilitate sending messages between processes. Additionally, MPI_Recv() is blocking, providing for both synchronous and asynchronous communication. As an example of a send operation, consider
MPI_Send(&variable, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);

Of course, each of the variable positions mean something to the send command. In this case, one floating point variable is to be sent to process with rank dest. By comparison, a receive command may have the syntax

MPI_Recv(&variable, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);

To summarize the capabilities of PVM and MPI in a nutshell, here are the most fundamental function calls in both environments:

PVM:

Start:  int pvm_mytid();
Send:  pvm_initsend(int encoding);
pvm_pk...(datatype *ptr, int size, int stride);
pvm_send(int task_id, int msgtag);
Receive:  pvm_recv(int task_id, int msgtag);
pvm_upk...(datatype *ptr, int size, int stride);
Shut down:  pvm_kill(int task_id);
pvm_exit();

MPI:

Start:  MPI_Init();
Send:  MPI_Send(void *message, int size,
MPI_Datatype datatype, int destination, int tag,
MPI_Comm communicator);
Receive:  MPI_Recv(void *message, int size,
MPI_Datatype datatype, int source, int tag,
MPI_Comm communicator, MPI_Status status);
Shut down:  MPI_Finalize();

MPI provides a wealth of commands to facilitate message passing (hence its name). Programming PVM and MPI is similar. If one learns to program in one of these environments it isn’t hard to learn to program in the other. Problem solving methodologies are similar in both programming systems. Students have been successful in programming MPI, since many of our students are experienced in C and C++ programming and are comfortable with C style function and library calls.

8 JAVA: Availability

Now it gets interesting. Java, with all of its devotees, has fast become the programming fad of the decade. Like any other hot new programming environment, Java has an ever-increasing following. A complete programming system, Java incorporates many tools to make it more than a general purpose programming system. Java tools may be obtained from many places on the World Wide Web and via anonymous ftp from several sites. The references to this paper contain several starting URL’s for searching for the desired programming tools. [7]

However, the interesting thing about Java is that it may be used for DPP, in a manner similar in some respects to the capabilities of PVM and MPI.

9 JAVA: Programming

Java programming is an object based programming system which was originally designed to facilitate programming for the World Wide Web. It has evolved into much more, and has found its way into general use, including, for example, teaching elementary programming concepts such as one might find in an introductory course at your favorite university. Anyone with experience in C++ will find that Java includes and embodies a similar programming style. Thus, the C++ programmer, with just a bit of effort, will find Java a relatively nice programming environment. However, in this discussion, we are concerned mainly with DPP.

Java has many features that make programming easier. However, they are outside the scope of this document. The features that make Java viable as a parallel programming language are:

• String support for networking.

• The ability to perform remote procedure calls and remotely register objects via the Remote Method Invocation (RMI) libraries.

• Built-in support for multi-threading.

• Java’s ability to invoke instances of the Java Virtual machine, dynamically load and link classes, and
invoke the compiler when necessary.[2]

Our primary goal is allowing one host to run programs on another host. Using the built-in threading capabilities of the language coupled with RMI, threads can be started on a slave workstation using a separate master workstation. In simplest terms, this happens by using RMI, passing a parameter to a remote method with a runnable class as a parameter, and then getting that runnable class to execute by invoking another remote method.

To understand how this works, we must discuss two of Java’s technologies: RMI and threading. In Java, there are a number of ways to go about creating a thread, but only two ways to create a threadable object. Since Java is truly object oriented, everything in the language is an object (except for primitives—but they can also be treated as objects). To create an object that is threadable, you either create a class that extends the Thread class or create a class that implements the interface Runnable. The second option is more desirable because Java doesn’t support multiple inheritance, so we are given the ability to extend some other class if necessary.

Once you are able to start a thread, then you will be able to export the thread to another machine using RMI. RMI gives you the ability to create remote objects, which you can pass around and call methods just like any other Java object. The unique thing about the remote object is that the object isn’t physically located on the host machine. You gain reference to these remote objects by using the Java based RMI Registry. On the remote computer, the code running must bind an object to the local RMI Registry using the Naming.rebind() method. However, before an object can be bound you must run the RMIC compiler on the bytecode of the object. This compiler generates two additional postprocessed classes: the Stub class and the Skeleton class.

Once this binding is done, then a remote computer can gain reference to this object by calling the Naming.lookup method. This lookup function contacts the remote registry at a specified IP or URL, requests the stub of the class, and then loads that stub class into the local Virtual Machine. Then, once you have the reference to that class, you can make method calls on this object like any other object. The only difference is that when you make a method call on the remote object, all the parameters are marshaled (prepared for communication), sent to the remote registry, unmarshaled (unpacked), executed, marshaled again, send back to the local machine, and unmarshaled yet again. As you see, this can be a very powerful component of your programming.

As you can imagine, these features present us with enormous opportunities for creating parallel or distributed computing applications. While the nitty-gritty programming of Java differs from programming PVM or MPI, the fundamental capabilities are similar.

10 Summary

Programming a distributed environment using a message passing scheme is straightforward using PVM or MPI. Programming a distributed cluster of workstations using JAVA is not yet as mature as PVM or MPI, and is limited to the JAVA structure. Source code for many different PVM, MPI, or Java programming applications is available upon request.

11 Bibliography

[1] Peter S. Pacheco, Parallel Programming with MPI, Morgan Kaufmann, 1997


[3] Ian Foster, Designing and Building Parallel Programs, Addison Wesley, 1994


