PVM & MPI in Parallel & Distributed Systems

Dr. P. Varaprasada Rao
Professor
Department of Computer Science & Engineering
Gokaraju Rangaraju Institute of Engineering & Technology Jawaharlal Nehru Technological University, Hyderabad, India, 501301

Abstract— *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). This research is aimed to explore each of these two vehicles for DPP, considering capability, ease of use, and availability, and compares their distinguishing features; and also explores programmer interface and their utilization for solving real world parallel processing applications. This work recommends a potential research issue, that is, to study the feasibility of creating a programming environment that allows access to the virtual machine features of PVM and the message passing features of MPI. PVM and MPI, two systems for programming clusters, are often compared. Each system has its unique strengths and this will remain so in to the foreseeable future. This paper compares PVM and MPI features, pointing out the situations where one may be favored over the other; it explains the deference's between these systems and the reasons for such deference's. The research conducted herein concludes that each API has its unique features of strength, hence has potential to remain active into the foreseeable future.*

Keywords— Message Passing Interface, Network of Workstations, Parallel Virtual Machine, Parallel and Distributed Processing

I. INTRODUCTION

Parallel processing, the method of having many small tasks solve one large problem, has emerged as a key enabling technology in modern computing. The past several years have witnessed an ever- increasing acceptance and adoption of parallel processing. The acceptance has been facilitated by two major developments: massively parallel processors (MPPs) and the widespread use of distributed computing. MPPs are probably the most powerful computers in the world. These machines combine a few hundred to a few thousand CPUs in a single large cabinet connected to hundreds of gigabytes of memory and over enormous computational power. But the cost of such machines is very high, they are very expensive. The second major development alerting scientific problem solving is distributed computing. Distributed computing is a process whereby a set of computers connected by a network are used collectively to solve a single large problem. The idea of using such clusters or networks of workstations to solve a parallel problem became very popular because such clusters allow people to take advantage of existing and mostly idle workstations and computers, enabling them to do parallel processing without having to purchase an expensive supercomputer. As more and more organizations have high-speed local area networks interconnecting many general-purpose workstations, the combined computational resources may even exceed the power of a single high-performance computer. That's why the cluster is considered to be the hottest trend today. Common between distributed computing and MPP is the notion of message passing. In all parallel processing, data must be exchanged between cooperating tasks. Message passing libraries have made it possible to map parallel algorithm onto parallel computing platform in a portable way. PVM and MPI have been the most successful of such libraries. Now PVM and MPI are the most used tools for parallel programming. Since there are freely available versions of each, users have a choice, and beginning users in particular can be confused by their superficial similarities. So it is rather important to compare these systems in order to understand under which situation one system of programming might be favored over another, when one is more preferable than another.

One of MPI's prime goals was to produce a system that would allow manufacturers of high-performance massively parallel processing (MPP) computers to provide highly optimized and efficient implementations. In contrast, PVM was designed primarily for networks of workstations, with the goal of portability, gained at the sacrifice of optimal performance. PVM has been ported successfully too many MPPs by its developers and by vendors, and several enhancements including in-place data packing and pack-send extensions have been implemented with much success. Nevertheless, PVM's inherent message structure has limited overall performance when compared with that of native communications systems.

A. Parallel Programming Fundamentals

1) Parallel machine model: cluster

Sequential Machine Model or single Machine Model, the von Neumann computer comprises a central processing unit (CPU) connected to a storage unit (memory). The CPU executes a stored program that specifies a sequence of read and writes operations on the memory. This simple model has proved remarkably robust [3]. Really programmers can be trained in the abstract art of \programming" rather than the craft of \programming machine X" and can design algorithms for an abstract von Neumann machine, confident that these algorithms will execute on most target computers with reasonable efficiency. Such machine is called SISD (Single Instruction Single Data) according to Flynn's taxonomy; it means that single instruction stream is serially applied to a single data set.

A cluster comprises a number of von Neumann computers, or nodes, linked by an interconnection network (see Figure 1). Each computer executes its own program. This program may access local memory and may send and receive messages over the network. Messages are used to communicate with other computers or, equivalently, to read and write remote memories. Such cluster is most similar to what is often called the distributed-memory MIMD (Multiple Instruction Multiple Data) computer. MIMD

means that each processor can execute a separate stream of instructions on its own local data; distributed memory means that memory is distributed among the processors, rather than placed in a central location.

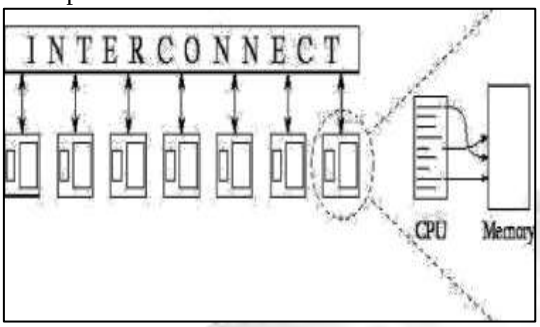

Fig. 1: The cluster. Each node consists of a von Neumann machine: a CPU and memory.

A node can communicate with other nodes by sending and receiving messages over an interconnection network.

2) Parallel programming model: message passing paradigm

The sequential paradigm for programming is a familiar one. The programmer has a simplified view of the target machine as a single processor which can access a certain amount of memory. He or she therefore writes a single program to run on that processor and the program or the underlying algorithm could in principle be ported to any sequential architecture. The message passing paradigm is a development of this idea for the purposes of parallel programming. Several instances of the sequential paradigm are considered together. That is, the programmer imagines several processors, each with its own memory space, and writes a program to run on each processor. Each processor in a message passing program runs a separate process (sub-program, task), and each such process encapsulates a sequential program and local memory (In effect, it is a virtual von Neumann machine). Processes execute concurrently. The number of processes can vary during program execution. Each process is identified by a unique name (rank) (see Figure 2). So far, so good, but parallel programming by definition requires cooperation between the processors to solve a task, which requires some means of communication. The main point of the message passing paradigm is that the processes communicate via special subroutine calls by sending each other message.

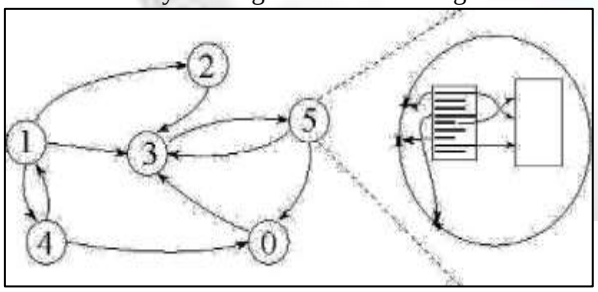

Fig. 2: The figure shows both the instantaneous state of a computation and a detailed picture of a single process (task). A computation consists of a set of processes. A process encapsulates a program and local memory. (In effect, it is a virtual von Neumann machine.)

B. PVM & MPI

Usually differences between systems for programming can be traced to explicit differences in the goals of the two systems, their origins, or the relationship between their specifications and implementations. That's why we prefer to analyze the differences in PVM and MPI by looking first at sources of these differences, it will help better illustrate how PVM and MPI differ and why each has features the other does not.

1) Background and goals of design

The development of PVM started in summer 1989 at Oak Ridge National Laboratory (ORNL). PVM was effort of a single research group, allowing it great flexibility in design and also enabling it to respond incrementally to the experiences of a large user community. Moreover, the implementation team was the same as the design team, so design and implementation could interact quickly. Central to the design of PVM was the notion of a "virtual machine" a set of heterogeneous hosts connected by a network that appears logically to user as a single large parallel computer or parallel virtual machine, hence its name. The research group, who developed PVM, tried to make PVM interface simple to use and understand. PVM was aimed at providing a portable heterogeneous environment for using clusters of machines using socket communications over TCP/IP as a parallel computer. Because of PVM's focus on socket based communication between loosely coupled systems, PVM places a great emphasis on providing a distributed computing environment and on handling communication failures. Portability was considered much more important than performance (for the reason that communications across the internet was slow); the research was focused on problems with scaling, fault tolerance and heterogeneity of the virtual machine [3]. The development of MPI started in April 1992. In contrast to the PVM, which evolved inside a research project, MPI was designed by the MPI Forum (a diverse collection of implementers, library writers, and end users) quite independently of any specific implementation, but with the expectation that all of the participating vendors would implement it. Hence, all functionality had to be negotiated among the users and a wide range of implementers, each of whom had a quite different implementation environment in mind. Some of these goals (and some of their implications) were the following [1][2]:

- MPI would be a library for writing application programs, not a distributed operating system.
- MPI would provide source-code portability.
- MPI would allow efficient implementation across a range of architectures.
- MPI would be capable of delivering high performance on high-performance systems. Scalability, combined with correctness, for collective operations required that group be "static".
- MPI would support heterogeneous computing, although it would not require that all implementations be heterogeneous (MPICH, LAM are implementations of MPI that can run on heterogeneous networks of workstation) MPI would require well-defined behavior.
- The MPI standard has been widely implemented and is used nearly everywhere, attesting to the extent to which these goals were achieved.

2) Definitions

a) What is MPI?

– MPI (Message Passing Interface) is specification for message-passing libraries that can be used for writing portable parallel programs.

b) What does MPI do?

– When we speak about parallel programming using MPI, we imply that:
– A fixed set of processes is created at program initialization; one process is created per processor
– Each process knows its personal number
– Each process knows number of all processes
– Each process can communicate with other processes
– Process can't create new processes; the group of processes is static.

c) What is PVM?

PVM (Parallel Virtual Machine) is a software package that allows a heterogeneous collection of workstations (host pool) to function as a single high performance parallel virtual machine. PVM, through its virtual machine, provides a simple yet useful distributed operating system. It has daemon running on all computers making up the virtual machine. PVM daemon (pvmd) is UNIX process, which oversees the operation of user processes within a PVM application and coordinates inter-machine PVM communications. Such pvmd serves as a message router and controller. One pvmd runs on each host of a virtual machine, the first pvmd, which is started by hand, is designated the master, while the others, started by the master, are called slaves. It means, that in contrast to MPI, where master and slaves start simultaneously, in PVM master must be started on our local machine and then it automatically starts daemons on all other machines. In PVM only the master can start new slaves and add them to configuration or delete slave hosts from the machine. Each daemon maintains a table of configuration and handles information relative to our parallel virtual machine. Processes communicate with each other through the daemons: they talk to their local daemon via the library interface routines, and local daemon then sends/receives messages to/from remote host daemons.

The user writes his application as a collection of cooperating processes (tasks) that can be performed independently in different processors. Processes access PVM/MPI resources through a library of standard interface routines. These routines allow the initiation and termination of processes across the network as well as communication between processes.

3) What is not different?

Despite their differences, PVM and MPI certainly have features in common. In this section we review some of the similarities.

a) Portability

Both PVM and MPI are portable; the specification of each is machine independent, and implementations are available for a wide variety of machines. Portability means, that source code written for one architecture can be copied to a second architecture, compiled and executed without modification.

b) MPMD

Both MPI and PVM permit different processes of a parallel program to execute different executable binary files (This would be required in a heterogeneous implementation, in any case). That is, both PVM and MPI support MPMD programs as well as SPMD programs, although again some implementation may not do so.

c) Interoperability

The next issue is interoperability - the ability of different implementations of the same specification to exchange messages. For both PVM and MPI, versions of the same implementation (Oak Ridge PVM, MPICH, or LAM) are interoperable.

d) Heterogeneity

The next important point is support for heterogeneity. When we wish to exploit a collection of networked computers, we may have to contend with several different types of heterogeneity.

e) Architecture

The set of computers available can include a wide range of architecture types such as PC class machines, high-performance workstations, shared memory multiprocessors, vector supercomputers, and even large MPPs. Each architecture type has its own optimal programming method. Even when the architectures are only serial workstations, there is still the problem of incompatible binary formats and the need to compile a parallel task on each different machine.

f) Data Format

Data formats on different computers are often incompatible. This incompatibility is an important point in distributed computing because data sent from one computer may be unreadable on the receiving computer. Message passing packages developed for heterogeneous environments must make sure all the computers understand the exchanged data; they must include enough information in the message to encode or decode it for any other computer.

g) Computational Speed

Even if the set of computers are all workstations with the same data format, there is still heterogeneity due to different computational speeds. The problem of computational speeds can be very subtle. The programmer must be careful that one workstation doesn't sit idle waiting for the next data from the other workstation before continuing.

h) Machine Load

Our cluster can be composed of a set of identical workstations. But since networked computers can have several other users on them running a variety of jobs, the machine load can vary dramatically. The result is that the effective computational power across identical workstations can vary by an order of magnitude.

i) Network Load

Like machine load, the time it takes to send a message over the network can vary depending on the network load imposed by all the other network users, who may not even be using any of the computers involved in our computation. This sending time becomes important when a task is sitting idle waiting for a message, and it is even more important when the parallel algorithm is sensitive to message arrival time. Thus, in distributed computing, heterogeneity can appear dynamically in even simple setups.

Both PVM and MPI provide support for heterogeneity.

j) Virtual topology

A virtual topology is a mechanism for naming the processes in a group in a way that fits the communication pattern better. The main aim of this is to make subsequent code simpler. It may also provide hints to the run-time system

which allow it to optimize the communication or even hint to the loader how to configure the processes.

k) Message passing operations

MPI is a much richer source of communication methods than PVM. PVM provides only simple message passing, whereas MPI1 specification has 128 functions for message-passing operations, and MPI 2 adds an additional 120 functions to functions specified in the MPI 1.

l) Fault Tolerance

Fault tolerance is a critical issue for any large scale scientific computer application. Long running simulations, which can take days or even weeks to execute, must be given some means to gracefully handle faults in the system or the application tasks. Without fault detection and recovery it is unlikely that such application will ever complete. For example, consider a large simulation running on dozens of workstations. If one of those many workstations should crash or be rebooted, then tasks critical to the application might disappear. Additionally, if the application hangs or fails, it may not be immediately obvious to the user. Many hours could be wasted before it is discovered that something has gone wrong. So, it is very essential that there be some well-defined scheme for identifying system and application faults and automatically responding to them, or at least providing timely notification to the user in the event of failure.

The problem with the MPI-1 model in terms of fault tolerance is that the tasks and hosts are considered to be static. An MPI-1 application must be started en masse as a single group of executing tasks. If a task or computing resource should fail, the entire MPI-1 application must fail. This is certainly effective in terms of preventing leftover or hung tasks. However, there is no way for an MPI program to gracefully handle a fault, let alone recover automatically. As we said before, the reasons for the static nature of MPI are based on performance.

MPI 2 includes a specification for spawning new processes. This expands the capabilities of the original static MPI-1. New processes can be created dynamically, but MPI-2 still has no mechanism to recover from the spontaneous loss of process.

PVM supports a basic fault notification scheme: it doesn't automatically recover an application after a crash, but it does provide polling and notification primitives to allow fault-tolerant applications to be built. Under the control of the user, tasks can register with PVM to be notified" when the status of the virtual machine changes or when a task fails. This notification comes in the form of special event messages that contain information about the particular event. A task can \post" a notify for any of the tasks from which it expects to receive a message. In this scenario, if a task dies, the receiving task will get a notify message in place of any expected message. The notify message allows the task an opportunity to respond to the fault without hanging or failing.

This type of virtual machine notification is also useful in controlling computing resources. The Virtual Machine is dynamically reconfigurable, and when a host exits from the virtual machine, tasks can utilize the notify messages to reconfigure themselves to the remaining resources. When a new host computer is added to the virtual machine, tasks can be notified of this as well. This information can be used to redistribute load or expand the computation to utilize the new resource.

C. COMPARISON CAPABILITIES OF PVM , MPI AND JAVA FOR DISTRIBUTED PARALLEL PROCESSING

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 began to make an impact in the DPP arena as well.

1) 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 heterogeneous cluster of workstations as well as many shared memory parallel processing machines.[4]

2) 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): 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:

```
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.

3) 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]

4) 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.

5) 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.

6) 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.

II. CONCLUSION

In this paper we compared the features of the two systems, PVM and MPI, and pointed out situations where one is better suited than the other. If an application is going to be developed and executed on a single MPP, where every

processor is exactly like every other in capability, resources, software, and communication speed, then MPI has the advantage of expected higher communication performance. MPI has a much richer set of communication functions, so MPI is favored when an application is structured to exploit special communication modes not available in PVM (The most often cited example is the non-blocking send). In contrast to PVM, MPI is available on all massively parallel supercomputer. Because PVM is built around the concept of a virtual machine, PVM is particularly effective for heterogeneous applications that exploit specific strengths of individual machines on a network. The larger the cluster of hosts or the time of program's execution, the more important PVM's fault tolerant features becomes, in this case PVM is considered to be better than MPI, because of the lack of ability to write fault tolerant application in MPI. The MP specification states that the only thing that is guaranteed after an MPI error is the ability to exit the program. Programmers should evaluate the functional requirements and running environment of their application and choose the system that has the features they need.

REFERENCES

[1] W. Gropp and E. Lusk. Goals Guiding Design: PVM and MPI.
[2] I. Foster. Designing and building parallel programs. Addison-Wesley, 1995. ISBN 0-201- 57594-9, http://www.mcs.anl.gov/dbpp.
[3] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features. Calculateurs Paralleles, 8(2), 1996.
[4] Peter S. Pacheco, Parallel Programming with MPI, Morgan Kaufmann, 1997.
[5] Elliotte Rusty Harold, Java Network Programming, O'Reilly, 1997.
[6] Al Geist et.al., PVM, Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing, The MIT Press, 1994.