International Journal of Communication Networks and Security

Volume 2 | Issue 3 Article 11

January 2014

COMPARATIVE ANALYSIS OF PVM AND MPI FOR THE DEVELOPMENT OF PHYSICAL APPLICATIONS IN PARALLEL AND DISTRIBUTED SYSTEMS

SRIKANTH. BETHU
Department of Computer Science and Engineering, Holy Mary Institute of Technology and Science, Hyderabad, India, srikanthbethu@gmail.com

COMPARATIVE ANALYSIS OF PVM AND MPI FOR THE DEVELOPMENT OF PHYSICAL APPLICATIONS IN PARALLEL AND DISTRIBUTED SYSTEMS

SRIKANTH. BETHU

Assistant Professor, Department of Computer Science and Engineering, Holy Mary Institute of Technology and Science, Hyderabad, India
Jawaharlal Nehru Technological University, Hyderabad, A.P, India
E-mail: srikanthbethu@gmail.com

Abstract- This research is aimed to explore each of these two Parallel Virtual Machine (PVM) and Message Passing Interface(MPI) vehicles for DPP (Distributed Parallel Processing) 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. The comparisons usually start with the unspoken assumption that PVM and MPI represent different solutions to the same problem. In this paper we show that, in fact, the two systems often are solving different problems. In cases where the problems do match but the solutions chosen by PVM and MPI are different, we explain the reasons for the differences. Usually such differences can be traced to explicit differences in the goals of the two systems, their origins, or the relationship between their specifications and their implementations. This paper also 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.

Keywords- *Message Passing Interface , 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. 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.

II. PARALLELPROGRAMMING FUDAMENTALS

i) 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.

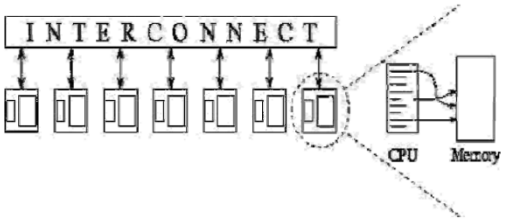

Figure 1: The cluster. Each node consists of a von Neumann machine: a CPU and memory. A node can communicate with other nodes by sending and receiving messages over an interconnection network.

a. PVM communication issues
   - Master/ Slave principle
   - TCP/IP socket communication
   - Loosely coupled and Tightly coupled

b. PVM performance issues
   - It provides Portability rather than Performance.

ii) Parallel programming model: Message passing

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.

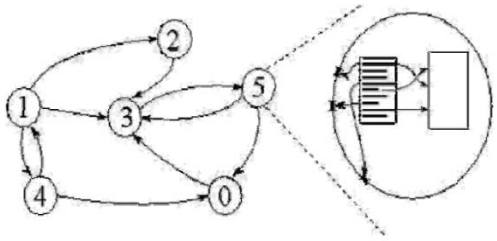

Figure 2: The figure shows both the instantaneous state of a computation and a detailed picture of a single process (task). A computation consists of a set of processes. A process encapsulates a program and local memory. (In effect, it is a virtual von Neumann machine.)

a. MPI communication issues
   - Through message passing between the processor into memory.
   - SPMD (Single Program Single Data) and MPMD (Multiple Program Multiple Data) functions.

b. MPI performance issues
   - It has the capability of delivering high performance on high performance systems with high scalability.

III. PVM AND 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.

a. 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.

MPI and its Goals:

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

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.

Some of these goals (and some of their implications) were the following [1][2]:

- 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.
- MPI would be a library for writing application programs, not a distributed operating system.

PVM and its Goals:

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.

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.

PVM had, with the exception of support for heterogeneous computing and a different approach to extensibility, different goals. In particular, 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 greater emphasis on providing a distributed computing environment and on handling communication failures.

Master/Slave principle in PVM:

The master/slave programming model is a very popular model used in distributed computing. In this model exists two separate programs, master and slave program. The master has the control over the running application, it controls all data and it calls the slave to

do their work. So the master is a separate "control" program, which is responsible for process spawning, initialization, collection and display of results. The slave programs perform the actual computation involved; they either are allocated their workloads by the master (statically or dynamically) or perform the allocations themselves.

b. PVM and MPI equal issues:

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

- 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.

- 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.

- 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.

- 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.

i. 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.

ii. 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.

iii. 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.

iv. 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.

v. 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.

c. PVM and MPI differences

- 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.

- 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.

- 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.

- Process Control

Process control refers to the ability to start and stop tasks, to find out which tasks are running, and possibly where they are running. PVM contains all of these capabilities; it can spawn/kill tasks dynamically. In contrast MPI - 1 has no defined method to start new task. MPI - 2 contains functions to start a group of tasks and to send a kill signal to a group of tasks.

- Resource control
  - In terms of resource management, PVM is inherently dynamic in nature. Computing resources or "hosts" can be added and deleted at will, either from a system "console" or even from within the user's application. Allowing applications to interact with and manipulate thei computing environment provides a powerful paradigm for
  - load balancing | when we want to reduce idle time for each machine involved in computation
  - task migration | user can request that certain tasks execute on machines with particular data formats, architectures, or even on an explicitly named machine

IV. IMPLEMENTATION AND COMPARISON

a) Portability, Heterogeneity, and Interoperability

Portability refers to the ability of the same source code to be compiled and run on different parallel machines. Heterogeneity refers to portability to “virtual parallel machines” made up of networks of machines that are physically quite different. Interoperability refers to the ability of different implementations of the same specification to exchange messages. In this section we compare PVM and MPI with respect to these three properties.

Portability is an underappreciated issue. PVM is considered by many to be highly portable, and in fact the PVM group has done an excellent job in providing implementations across a wide range of platforms, covering most Unix systems and Windows [24]. But the designers of MPI had to consider running on systems that were neither; in fact, MPI has even been used in embedded systems (see http://www.mc.com). MPI could not assume that any particular operating system support was available; the

design of MPI reflects this constraint. Some users have complained that MPI does not mandate support for certain Unix features, when in fact features such as standard input, process creation, and signals are absent in many important, non-Unix systems.

Support for Heterogeneity is provided in both specifications. PVM has separate functions to pack specific data types into buffers; MPI uses basic and derived data types. The MPI specification does not mandate heterogeneous support, however; that is up to the implementation. LAM [2], CHimP [1], and MPICH [3] are implementations of MPI that can run on heterogeneous networks of workstations.

Interoperability is outside the scope of the user program, and entirely up to the implementation. Some vendor implementations of PVM are neither heterogeneous nor interoperable with the Oak Ridge version of PVM. The MPI standard does not mandate implementation details, and thus MPI implementations, of which there are many, typically are not interoperable. Thus, "interoperability" of MPI matches that of PVM. Versions of the same implementation (Oak Ridge PVM, MPICH, or LAM) are interoperable. True interoperability is among completely different implementations, matched at the level of the wire protocol. A separate effort (not part of the MPI Forum) has developed an "interoperability standard" called IMPI that provides sufficient standardization for some implementations details so that implementations conforming to this standard can exchange messages.

When we compare implementations rather than an implementation of PVM with the MPI standard, the gap in this type of functionality narrows. For example, MPICH [3], rather than MPI, does provide a way for debuggers like Total view to access to internal MPICH state on the message queues. Many users want this information, but it raises an interesting issue: How does one define a standard for the internal state of an implementation? For any implementation this can be done, but different implementations may have different internal states. For example, one optimization for communication has the process issuing an MPI RECV send a message to the expected source of the message, allowing the sender to deliver the message directly into the receiver's memory [2]. Should this information be presented to the user? Other implementation choices might eliminate some queues altogether or makes it more difficult to find all pending communication operations; in fact, in the MPICH implementation, there is no send queue unless the system has been configured and built to support the message queue service. By not specifying a model of the internals of an MPI implementation, such as defining a "message queue" does, the MPI standard allows MPI implementations to make tradeoffs between the performance and functionality that the users want.

V. 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] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. Parallel Computing, 27(11):1479–1495, Oct. 2001.