Comparative analysis of PVM and MPI for the development of physical applications on parallel clusters*

Ekaterina Elts
scientific adviser: Assoc. Prof. A. V. Komolkin

Faculty of Physics, Saint-Petersburg State University, Russia
JASS 2004, St. Petersburg

Abstract

PVM and MPI, two systems for programming clusters, are often compared. Each system has its unique strengths and this will remain so into 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 differences between these systems and the reasons for such differences.

*PVM – Parallel Virtual Machine; MPI – Message Passing Interface

Contents

1 Introduction **3**

2 Parallel programming fundamentals **3**
2.1 Parallel machine model: cluster . . . 3
2.2 Parallel programming model: message–passing paradigm . . . 4
2.3 SPMD and MPMD . . . 5
2.4 Master/Slave principle . . . 6

3 PVM and MPI **6**
3.1 Background and goals of design . . . 6
3.2 Definitions . . . 7
3.3 What is not different? . . . 8
3.3.1 Portability . . . 8
3.3.2 MPMD . . . 8
3.3.3 Interoperability . . . 8
3.3.4 Heterogeneity . . . 8
3.4 Differences . . . 9
3.4.1 Process Control . . . 9
3.4.2 Resource Control . . . 10
3.4.3 Virtual Topology . . . 10
3.4.4 Message Passing operations . . . 11
3.4.5 Fault Tolerance . . . 11

4 Conclusion **12**

1 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 offer enormous computational power. But the cost of such machines is very high, they are very expensive.

The second major development affecting 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 on is more preferable than another.

The structure of this paper is as follows. In Section 2 we review the parallel programming fundamentals: the parallel machine model (cluster) (2.1) and the parallel programming model, which is used by PVM and MPI, – message passing paradigm (2.2), and then we give the definitions of SPMD and MPMD models (2.3) and Master/Slave principle (2.4). Section 3 is devoted to PVM and MPI systems. We consider the goals and history of the design of these systems (3.1), give the definitions of PVM and MPI (3.2), discuss their features and carry out the comparative analysis between PVM and MPI (3.3 and 3.4).

2 Parallel programming fundamentals

2.1 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 write operations on the memory.

This simple model has proved remarkably robust [Fos95]. 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

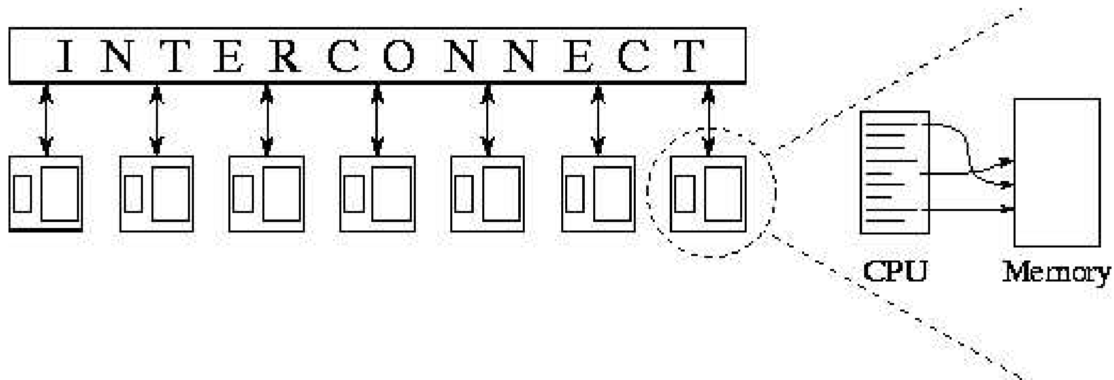

Figure 1: The cluster. Each node consists of a von Neumann machine: a CPU and memory. A node can communicate with other nodes by sending and receiving messages over an interconnection network.

Instruction Single Data) according to Flynn's taxonomy [1] , 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.

2.2 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 messages.

[1]Flynn's Taxonomy – A classification system for architectures that has two axes: the number of instructions streams executing concurrently, and the number of data sets to which those instructions are being applied. The scheme was proposed by Flynn in 1966.

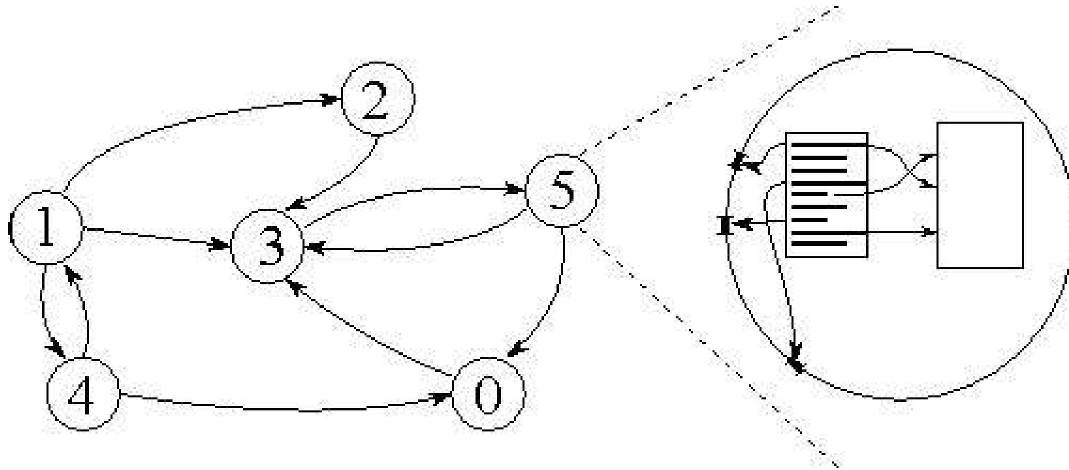

Figure 2: The figure shows both the instantaneous state of a computation and a detailed picture of a single process (task). A computation consists of a set of processes. A process encapsulates a program and local memory. (In effect, it is a virtual von Neumann machine.)

Messages are packets of data moving between processes. A message transfer is when data moves from variables in one sub-program to variables in another sub-program. The message passing system has no interest in the value of this data. It is only concerned with moving it. In general the following information has to be provided to the message passing system to specify the message transfer:

- Which processor is sending the message.
- Where is the data on the sending processor.
- What kind of data is being sent.
- How much data is there.
- Which processor(s) are receiving the message.
- Where should the data be left on the receiving processor.
- How much data is the receiving processor prepared to accept

In general the sending and receiving processors will cooperate in providing this information. Some of this information provided by the sending processor will be attached to the message as it travels through the system and the message passing system may make some of this information available to the receiving processor.

2.3 SPMD and MPMD

The basic idea of parallel computing, that an application consists of several processes executing concurrently. Each process is responsible for a part of the application's computational workload.

Sometimes an application is parallelized along its functions; each process performs a different function, for example, input, problem setup, solution, output, and display. Such model is called MPMD (Multiple Program Multiple Data).

A more common model of parallelizing an application is called SPMD (Single Program Multiple Data). In this method all the processes are the same, but each one only knows and solves a small part of data. So, each process executes the same program but operates on different data.

2.4 Master/Slave principle

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 there 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.

3 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[2] [GL97, GL]. 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.

3.1 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 greater 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 [GBD+94].

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 implementors, 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

[2] One common confusion in comparing MPI with PVM comes from comparing the specification of MPI with the implementation of PVM. Standards specifications tend to specify the minimum level of compliance, while any implementation offers more functionality.

to be negotiated among the users and a wide range of implementors, each of whom had a quite different implementation environment in mind.

The first task of the MPI Forum was to define the goals for MPI specification; some of these goals (and some of their implications) were the following [GL97, GL]:

- MPI would be a library for writing application programs, not a distributed operating system. This goal has implications for resource management issues, as discussed later
- 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.

3.2 Definitions

What is MPI?

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 (in MPI–1), the group of processes is static

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.

General idea of using MPI and PVM is the following:
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.

3.3 What is not different?

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

3.3.1 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.

3.3.2 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 (MPICH, LAM – support).

3.3.3 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.

3.3.4 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 [GBD+94]:

- 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.

- 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.

- 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.

- 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.

- 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.

As for MPI, different datatypes can be encapsulated in a single derived type, thereby allowing communication of heterogeneous messages. In addition, data can be sent from one architecture to another with data conversion in heterogeneous networks (big-endian, little-endian). Although MPI specification is designed to encourage heterogeneous implementation, some implementations of MPI may not be used in a heterogeneous environment. Both the MPICH and LAM are implementations of MPI, which support heterogeneous environments.

The PVM system supports heterogeneity in terms of machines, networks, and applications. With regard to message passing, PVM permits messages containing more than one datatype to be exchanged between machines having different data representations.

In summary, both PVM and MPI are systems designed to provide users with libraries for writing portable, heterogeneous, MPMD programs.

3.4 Differences

PVM is built around the concept of a virtual machine which is a dynamic collection of (potentially heterogeneous) computational resources managed as a single parallel computer. The virtual machine concept is fundamental to the PVM perspective and provides the basis for heterogeneity, portability, and encapsulation of function that constitute PVM.

In contrast, MPI has focused on message-passing and explicitly states that resource management and the concept of a virtual machine are outside the scope of the MPI (1 and 2) standard [GKP96].

3.4.1 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 [NS02].

3.4.2 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 their 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
- fault tolerance

Another aspect of virtual machine dynamics relates to efficiency. User applications can exhibit potentially changing computational needs over the course of their execution. For example, consider a typical application which begins and ends with primarily serial computations, but contains several phases of heavy parallel computation. PVM provides flexible control over the amount of computational power being utilized. Additional hosts can be added just for those portions when we need them.

MPI lacks such dynamics and is, in fact, specifically designed to be static in nature to improve performance. Because all MPI tasks are always present, there is no need for any time-consuming lookups for group membership. Each task already knows about every other task, and all communications can be made without the explicit need for a special daemon. Because all potential communication paths are known at startup, messages can also, where possible, be directly routed over custom task-to-task channels.

3.4.3 Virtual Topology

On the other hand, although MPI does not have a concept of a virtual machine, MPI does provide a higher level of abstraction on top of the computing resources in terms of the message-passing topology. In MPI a group of tasks can be arranged in a specific logical interconnection topology [NS02, For94] .

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.

For example, if our processes will communicate mainly with nearest neighbours after the fashion of a two-dimensional grid (see Figure 3), we could create a virtual topology to reflect this fact. What we gain from this creation is access to convenient routines which, for example, compute the rank of any process given its coordinates in the grid, taking proper account of boundary conditions. In particular, there are routines to compute the ranks of our nearest neighbours. The rank can then be used as an argument to message–passing operations.

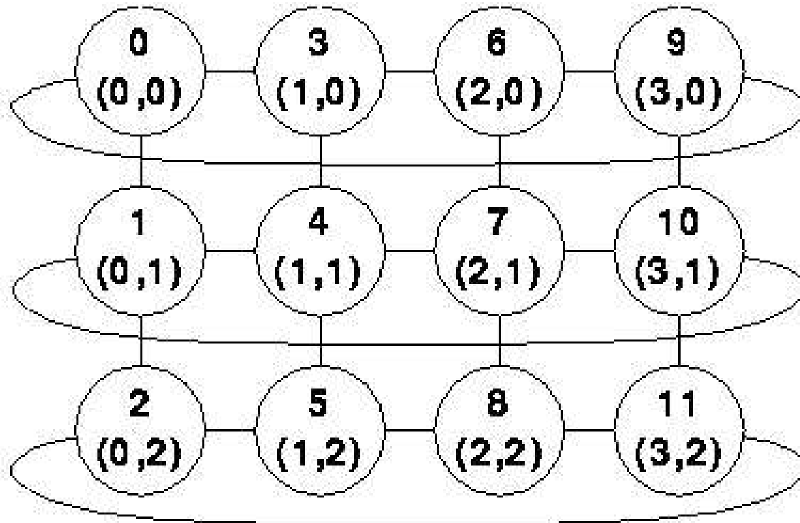

Figure 3: A virtual topology of twelve processes. The lines denote the main communication patterns, namely between neighbours. This grid actually has a cyclic boundary condition in one direction e.g. processes 0 and 9 are “connected”. The numbers represent the ranks and the conceptual coordinates mapped to the ranks.

Such cartesian virtual topologies, are suitable for grid-like topologies (with or without cyclic boundaries), in which each process is ”connected” to its neighbours in a virtual grid. MPI also allows completely general graph virtual topologies, in which a process may be ”connected” to any number of other processes and the numbering is arbitrary. These are used in a similar way to cartesian topologies, although of course there is no concept of coordinates.

3.4.4 Message Passing operations

MPI is a much richer source of communication methods than PVM. PVM provides only simple message passing, whereas MPI–1 specification has 128 functions for message-passing operations, and MPI–2 adds an additional 120 functions to functions specified in the MPI–1 [GKP96, For94].

3.4.5 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 [GKP96, For94].

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.

4 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 MPI 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

[For94] Message Passing Interface Forum. MPI: A Message–Passing Interface standard. *International Journal of Supercomputer Application*, 8(3/4):165–416, 1994. http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html.

[Fos95] I. Foster. *Designing and building parallel programs.* Addison-Wesley, 1995. ISBN 0-201-57594-9, http://www.mcs.anl.gov/dbpp.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, Cambridge, Mass., 1994. http://wwww.netlib.org/pvm3/book/pvm-book.html.

[GKP96] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8(2), 1996.

[GL97] W. D. Gropp and E. Lusk. Why are PVM and MPI so different? In J. Dongarra M. Bubak and J. Wasniewski, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 1332 of Lecture Notes in Computer Science*, pages 3–10. Springer Verlag, 1997. 4th European PVM/MPI User's Group Meeting, Cracow, Poland, November 1997.

[GL] W. Gropp and E. Lusk. Goals Guiding Design: PVM and MPI.

[NS02] S. Nemnugin and O. Stesik. *Parallel programming for multiprocessor computational systems.* BHV-Petersburg, 2002. - 400p, ISBN 5-94157-188-7 (In Russian).