Chapter 5

Review on Parallel Programming Tools
Chapter 5: Review on Parallel Programming Tools

5.1 Introduction

The demand of computational power in Computer Science is one of the most important aspects from the origin of computer systems. No doubt technology has picked its highest progress in processing power, data storage capacity, circuit integration scale etc. in last several few years but still it becomes unsatisfactory for some scientific computations for today’s applications. So trends for developing the high speed computer become one of the thrust areas of research. As present applications need huge computation power, there must be a solution with low cost and high desirable performance. A supercomputer is one of its solutions but it is of very very high cost and may be out of budget for many organizations. So parallel computing came into existence and is a very successful way of increasing desirable computation speed. A collection of workstations can be the computationally equivalent to a super computer. The computer networks become the ideal platforms for the parallel computing recently and this type of computations is known as network or heterogeneous computing [102].

Distributed heterogeneous computing may be defined as a particular form of parallel computing in which each computing task is processed on the most appropriate computing framework available. A large task consists several numbers of small manageable tasks which are concurrently executed on the most suitable framework to increase the efficiency. The degree of parallelism depends upon the parallel computer architecture.

Network computing interconnects different heterogeneous systems into a single unified computing resource [75, 103]. The network which is designed for the high computation should have communication potential that equivalent to the logical computational model of the application [103, 104]. The network designed for the purpose of high computing performance should involve the following criteria: scalable cumulative power [102, 103], decreased inter process communications, prerequisite for multiple logical communication channels with guaranteed bandwidths, support for flexible application-dependent virtual topologies [83, 84, 103, 104], support of easy partition, effective system management.

71
The benefit of parallel computing system is that it ensures the processing power of widely available various type of computer systems connected through networks because these networks are at leisure or partially idle most of the time. These are beautiful resource for efficient free processing power.

The available software tools for a high performance computing environment required the knowledge of internal architecture and component of a distributed system [105]. The communication and synchronization among the various users is the most important issue for the parallel computing environments. Therefore, over the last few years, a number of application development and representing tools, designed to solve the message passing problems in distributed environment. The communication tools implement one of the two communication techniques: message passing or shared memory. Message passing interface (MPI) and parallel virtual machine (PVM) are the most popular examples of the message passing tools [106]. So many parallel computing environments are available to support parallel execution of programs on computer networks. But which environment will be the best one? The straight forward answer of this question is very difficult. To determine which environment will be the best depends one so many factors like particular algorithm at hand, programmer’s proficiency as well as personal interests, ease of use, fluency and efficiency. The runtime-efficiency is only a single important factor which determines which environment would be the best one in general [107]. However, efficient utilization of the computational potential of computer networks is still an open problem and a research challenge in future.

Numerous programming frameworks like CPS, Linda, P4, POSYBL, PVM, Express, MPI, HeNCE have been developed to support parallel execution of programs on computer networks. These frameworks function at various levels of abstraction, represent different formal models for parallelism, exploit dedicated or general-purpose languages, and vary from local area network-based systems to geographically distributed systems. Each framework is developed on ad-hoc basis and thus do not cover all the issues that are found in heterogeneous computing [102]. So each framework is specialized for a particular type of application.
Different frameworks have different advantages as well as shortcomings. It is unrealistic that a framework that would be unanimously employed for any parallel distributed computing application. The environments are developed depending upon the requirements to solve the problem at current hand. But PVM becomes the de-facto environment for this purpose [108].

5.2 Classification of Simulation Tools

The simulation tools for parallel and distributed systems have been classified as shown by Fig. 5.1 [109]. These tools are very recently developed and most of them still are under research.

![Simulation Tools Diagram]

**Figure 5.1** Different Simulation Tools

5.3 Tools for Simulation of Parallel and Distributed Systems

This section provides an overview of some of the many available tools that simulate parallel and distributed systems. Table 5.1 describes the different simulation tools for parallel and distributed computing systems.
### Table 5.1 A Comparison Of Available Simulation Tools for Parallel and Distributed Systems

<table>
<thead>
<tr>
<th>No</th>
<th>Tool</th>
<th>Description</th>
<th>Developer</th>
<th>Target Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SimJava</td>
<td>SimJava [110] provides process based discrete event simulation with animation facilities through a collection of entities communicating with each other.</td>
<td>University of Edinburgh, UK</td>
<td>Discrete event simulation</td>
</tr>
<tr>
<td>2</td>
<td>Dimemas</td>
<td>Dimemas [111] is a performance analysis tool for message passing programs. It simulates the time behaviour of a parallel application on a machine or a cluster of machines modelled by a set of performance parameters.</td>
<td>The Technical University of Catalonia, Spain</td>
<td>Parallel program simulation</td>
</tr>
<tr>
<td>3</td>
<td>SvPablo</td>
<td>SvPablo [112] is a performance analysis tool that captures and analyses data from serial and parallel programs.</td>
<td>University of Illinois at Urbana-Champign, USA</td>
<td>Parallel program simulation</td>
</tr>
<tr>
<td>4</td>
<td>MPISim</td>
<td>MPISim [113] is a parallel simulator for parallel programs using Message Passing Interface (MPI).</td>
<td>University of California at Los Angeles, USA</td>
<td>Parallel program simulation</td>
</tr>
<tr>
<td>5</td>
<td>Pamela</td>
<td>Pamela [114] is a process-oriented performance simulation language that models the interaction of the parallel program and the machine.</td>
<td>Delft University of Technology, Netherlands</td>
<td>Parallel program simulation</td>
</tr>
<tr>
<td>6</td>
<td>Clue</td>
<td>Clue [115] is a simulation and performance assessment tool for parallel programs using message passing libraries for communication.</td>
<td>Vienna University of Technology, Austria</td>
<td>Parallel program simulation</td>
</tr>
<tr>
<td>7</td>
<td>Ptolemy 2</td>
<td>Ptolemy 2 [116] is a Java-based library that supports heterogeneous, concurrent modelling and design.</td>
<td>University of California at Berkeley, USA</td>
<td>Concurrent System simulation</td>
</tr>
<tr>
<td>8</td>
<td>Pepa</td>
<td>Pepa [117] known as Performance Evaluation Process Algebra is used for modelling concurrent systems which cooperate and share work tasks.</td>
<td>University of Edinburgh, UK</td>
<td>Concurrent system simulation</td>
</tr>
</tbody>
</table>
5.4 Parallel Programming Environments

So many parallel programming environments have been developed but some of them in brief are studied below.

5.4.1 CPS

CPS is a software package for splitting of computational task into jobs which are distributed over one or more processors. It was developed to function as a tool for solving computing problems which require many computing cycles per I/O byte and supports the RISC processor farms, operating in parallel [39]. CPS supports routines for message passing, remote procedure calls and process synchronization. Besides these functions, it provides bulk data transfers between cooperating processes and batch processing queues which are distinctive features of CPS [107].

5.4.2 LINDA

In computer science, Linda is a model of coordination and communication among several parallel processes operating upon objects stored in and retrieved from shared, virtual, associative memory [37, 38, 107, 118]. The combination of C with Linda gives the C-Linda programming language. This model is implemented as a "coordination language" in which several primitives operating on ordered sequence of typed data objects, "tuples," are added to a sequential language, such as C, and a logically global associative memory, called a tuplespace, in which processes store and retrieve tuples.

The original Linda model requires four operations that individual workers perform on the tuples and the tuplespace: in atomically reads and removes—consumes—a tuple from tuplespace, rd non-destructively reads a tuplespace, out produces a tuple, writing it into tuplespace, eval creates new processes to evaluate tuples, writing the result into tuple space [37, 38, 118].

Compared to other parallel-processing models, Linda is more orthogonal in treating process coordination as a separate activity from computation and it is more general in being able to subsume various levels of concurrency—uniprocessor, multi-threaded
multiprocessor, or networked—under a single model. It orthogonally allows processes computing in different languages and platforms to interoperate using the same primitives. Its generality allows a multi-threaded Linda system to be distributed across multiple computers, or vice-versa, without change [37, 118].

5.4.3 p4

Argonne National Laboratory has developed p4 which is a library of macros and subroutines for the purpose of programming a variety of parallel machines in C and Fortran. p4 supports computing across the cluster of shared memory computers [38, 82]. p4 is a portable software, simple to install and efficient in nature [38, 82]. It is used to program networks of workstations, distributed-memory parallel supercomputers like the Intel Paragon, the Thinking Machines CM-5 and the IBM SP-1 as well as shared-memory multiprocessors like the Kendall Square. It has currently been installed on the following list of machines: Sequent Symmetry (Dynix and PTX), Convex, Encore Multimax, Alliant FX/8, FX/800, and FX/2800, Cray X/MP, Sun (SunOS and Solaris), NeXT, DEC, Silicon Graphics, HP, and IBM RS6000 workstations, Stardent Titan, BBN GP-1000 and TC-2000, Kendall Square, nCube, Intel IPSC/860, Intel Touchstone Delta, Intel Paragon, Alliant Campus, Thinking Machines' CM-5 and the IBM SP-1 (TCP/Ethernet, TCP/switch, EUI, and EUI-H [38, 82].

5.4.4 HeNCE

HeNCE (Heterogeneous Network Computing Environment) is an X-window based software platform designed to support scientists in developing parallel programs that run on a computer network [38, 119]. High level abstraction for specifying parallelism is provided by this tool. An application program on a parallel programming paradigm is described by a graph in this tool. This HeNCE graphs may be in different variation of directed acyclic graphs or DAGS. Nodes of the graph represent subroutines and the arcs represent data dependencies. Individual nodes are executed under PVM. HeNCE is the collection of integrated graphical tools for creating, compiling, executing, and analyzing HeNCE programs [38, 119]. HeNCE trusts on the PVM system for process initialization.
and communication. HeNCE shows an event-ordered animation of application execution, visualization comparative computational speeds, processor utilization and load imbalances [38, 119].

5.4.5 PVM

Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) [120, 121, 122] are the most familiar examples of the message passing systems. PVM is particularly effective for heterogeneous applications that exploit specific strengths of individual machines on a network. The individual computers may be shared or local-memory multiprocessors, vector supercomputers, specialized graphics engines or scalar workstations that may be interconnected by a variety of networks such as Ethernet, FDDI etc. PVM is an integrated software tools and libraries that is mainly designed towards networks of workstations. The central notion to the design of PVM is virtual machine concept. Virtual machine is defined as the collection of heterogeneous computers connected by a network which appears to a user as a single large computation system [122]. So using the combined speed and storage of many computers, the large computational problem can be solved with more cost effectively. The PVM system has been used for applications such as molecular dynamics simulations, superconductivity studies, distributed fractal computations, matrix algorithms, and in the classroom as the basis for teaching concurrent computing. The PVM system consists of two parts. The first part is a daemon which is known as pvmd3 and simply known as pvmd. Pvm exists in all the computers making up virtual machine. Any user can install pvmd on a machine using a valid login [108]. A user willing to utilize PVM, first make an arrangement for a virtual machine by specifying a host-pool list. The daemons are started on each machine and co-operate to imitate a virtual machine. A machine to be a member of virtual machines, it must run its own daemon. The PVM application can then be started from a command line prompt on any of these machines [108, 123].
Multiple users can build up overlapping virtual machines and each user can run several PVM applications on simultaneous basis. The PVM applications co-ordinate with the daemons via sockets and / or pipes [124]. Thus applications can be signed up into PVM and then can be monitored by it although they were not started by it [124]. Applications are free to join or leave a virtual machine at any number of times allowing them to belong several virtual machines.

PVM supports dynamic process management while in other systems the processes are statistically defined [108]. Dynamic process groups are layered above the core PVM routines. A process can live to multiple groups, and groups can change dynamically at any time during a computation. Routines are provided for tasks to join and leave a named group. Group members are uniquely numbered from zero to the number of group members minus one. If gaps appear in this numbering due to tasks leaving the group, PVM attempts to fill these gaps with subsequently joining tasks. Tasks can also query for information about other group members. Functions that logically deal with groups of tasks such as broadcast and barrier use the users explicitly defined group names as arguments [108].
The second part of the system is a library of PVM interface routines [108, 123]. This library holds the functionally complete user callable routine for message passing, spawning process, co-coordinating tasks and modifying virtual machines. Application programs for the execution must be linked to these libraries like other programming languages. Moreover, PVM programs written for different architectures can communicate to each other, thus allowing for building of heterogeneous network computing systems. The PVM software contains a collection of protocol algorithms to implement reliable and sequenced data transfer, distributed consensus and mutual exclusion. These algorithms make the system robust by introducing error detection mechanisms and failure notification to applications. PVM uses both UDP and TCP sockets. UDP sockets are set up between a pair of daemons and between a daemon and a local task. The daemon-daemon socket is used for carrying data and inter-daemon control. When a user starts a daemon, the daemon sets up a single TCP socket with each daemon in virtual machine. These TCP carry standard-out and standard-error messages back to the user [125]. PVM requires a three-step procedure for a task to send (or receive) a message. For sending a message, a send buffer has first to be initiated, then the data is packed into the buffer, and finally the data in the buffer is sent. Later versions e.g. (PVM 3.3) provide the option to send data using a single call [126]. As the PVM becomes a de-facto software across the globe, it becomes mandatory to keep the PVM API backward compatible so that all existing PVM applications would continue to run as it is with newer versions.

![Figure 5.3 PVM Architectural View](image-url)
5.4.6 MPI

MPI is a common message passing library approach in which a process calls the library for the purpose of exchanging messages with another processes. MPI is a standardized interface for inter-process communication. The reason behind the design of MPI was that every Massively Parallel Processor (MPP) merchant was developing their own specific message-passing API. This is why a portable parallel application could not be written. To remove this serious problem all the vendors of MPP come to a single platform and MPI is the outcome of that. So each MPP vendor accept MPI as the standard message-passing API and as a result MPI becomes faster than PVM on MPP hosts as each vendor focuses to increase the performance of MPI [122].

The early versions of MPI provided only message passing primitives, MPI comes in a form of software library (e.g., MPICH), so the programmer can use calls to library functions to perform process management or data exchange between processes. Implementations of MPI exist for popular programming languages (e.g., Fortran, C, C++, and Java) on a variety of platforms including networks of workstations. The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few language-specific features. The earlier version MPI-1 has the salient features like: a huge numbers of point-to-point communication primitives, a huge numbers of collective communication routines among group of processors, a communication context that supports the design of safe parallel software libraries, ability to adjust with communication topologies, ability to generate derived data types which depict messages of non-contiguous data [122].

But application programs in MPI-1 were not portable across the network as there was no standard way to start MPI tasks on different nodes. In MPI-2 version the standard process creation and start-up functions are included. The following communications functions are included in MPI-2: non-blocking collective communication functions, language binding for C++, MPI_SPAWN functions to start both MPI and non-MPI processes, one-sided communication functions as put and get.
MPI-2 is a richer source of communication functions than PVM having a total of 248 functions while MPI-1 has 128 functions only [122].

5.5 Comparisons between PVM and MPI

Now we discuss the major differences between PVM and MPI in the following sections with respect to the following factors.

5.5.1 Portability

Portability refers to the ability of the same source code which is to be copied, compiled and executed on different platforms without modifications. MPI is portable but PVM is highly portable [122, 127]. MPI is portable in the sense that MPI applications as a whole run on any single architecture and it does not able to co-operate among the different processes, running on different architectures. PVM group has done excellent work to facilitate implementations across a wide range of architectures encompassing most UNIX systems and Windows [122, 127, 128].

5.5.2 Heterogeneity

Heterogeneity refers to portability to virtual parallel machines which are obviously of different architectures. Both specifications support heterogeneity. But MPI does not mandate of that [122, 129] though it depends on the type of implementations. This has a great advantage in the sense that nobody from any other vendor is allowed to use the machines of a vendor which otherwise may slowdown the systems of later one. PVM has specific functions for the support to heterogeneity. LAM [27], CHimP [130], and MPICH [131] are implementations of MPI that can run on heterogeneous networks of workstations [127].

5.5.3 Interoperability

Interoperability refers to the ability of different implementations of the same specification to exchange messages [127]. Since MPI does not mandate heterogeneity, there is no
question of interoperability. PVM application programs can run across any set of
different architectures and the processes can co-operate to exchange the information
without any problem. PVM programs are more flexible in this sense. 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. IMPI is now
available [132] and several vendor implementations exist [127].

Due to the lack of interoperability MPI always need to check the destination of every
message whether the message is for the same host or for the other hosts. If it is for other
host, the message must be converted into a format that is compatible for the other MPI
version [122]. Furthermore MPI implementation uses native communication functions
provided by architecture directly while PVM implementation use native communication
functions during the local communication or to another host with identical architecture.
But PVM uses standard communication functions for heterogeneous communications.

MPI and PVM also differ in language operability. In case of PVM, a program written in
FORTRAN can send a message which can be received by a program written in C and
vice-versa. But in MPI, a program written in FORTRAN does not feel to communicate
with a program written in C in spite of executing on the same architecture. These two
languages have two different interfaces and hence MPI does not compel two languages to
interoperate [122].

5.5.4 Virtual Machine

Virtual machine is defined as the dynamic collection of heterogeneous distributed
computers such as different workstations, personal computers and massively parallel
computers etc, connected by a network which appears to a user as a single large
computing machine [122, 124]. Virtual machine concept has brought a revolutionary
change in parallel distributed computing. PVM is totally fabricated around the virtual
machine concept. Virtual machine concept is the most fundamental feature of PVM
[122]. This feature is the foundation for providing the facilities like portability,
heterogeneity, interoperability and encapsulation of functions. On the other hand, MPI
Chapter 5: Review on Parallel Programming Tools

imposes attention towards message-passing and resource management and the virtual machine concept does not exist in it.

5.5.5 Topology

A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware. Although MPI does not support virtual machine concept, it supports a higher level abstraction on top of the computing resources in terms of message passing topology. The MPI group of tasks can be arranged in a specific logical interconnection topology. Tasks then communicate to each other within that topology. The underlying physical network topology supports with a considerable speed for message passing [122].

5.5.6 Dynamic Process Group

Sometimes we want to perform global operations like broadcasting a message only to a subset of all the processes. MPI allows us to define a subset of these processes in run time using MPI library calls. Processes in this group is numbered from 0 to n-1 where n is the number of a processor in a group [124]. Each process ID in a group is known as rank. User processes in MPI can create new processes at runtime.

5.5.7 Contexts

A context is a system defined tag. Every message has its own tag which can be used to distinguish messages from one another. So messages of different groups never mixed when we create multiple groups with overlapping processes. Both PVM and MPI supports context in different ways. In PVM, any task can send a request to any other tasks without considering the willingness of the receiving tasks. But in MPI, a task can send a message to specific tasks which are interested to receive that message due to the presence of separate message contexts for each task [133]. PVM 3.4 has a concept of base context.
In PVM all spawned tasks inherits their parent context. But if parent of any task dies, the child tasks inherit base context [122].

5.5.8 Communicators

The notions of group of processes and context are combined in a single object called a communicator [122]. Most communications are specified in terms of rank of the process in the group identified with the given communicator. Communicator variables are associated with the newly created groups in order to refer that group later as per the requirement. All communications takes place within a communicator. This has a great advantage in the sense that it provides high level of protection against irrelevant messages [124]. Communicator is the most essential part of MPI. PVM 3.4 would have been included communicators.

5.5.9 Process Control

Process control refers to the ability to begin and end processes, for finding out which processes are running, and where they are possibly running [122, 123]. PVM functions provide the ability to join or leave the virtual machines, to kill a process, to send a signal to a process, to check whether a process is running and to notify an arbitrary process if another leaves from PVM system.

PVM has some basic functions which are required to know how many processes can be started on the available computing resource. In the other hand MPI-1 has no defined functions that can start a parallel application. But MPI-2 is incorporated with some functions that can start a group of tasks and to send a kill signal to group of tasks [134].

5.5.10 Resource Management

PVM that is inherently dynamic in nature, can add or remove computing resources at will either from system console or even from within the user's applications. PVM permits applications to interact with and manipulates the computing environment to provide a powerful paradigm for load balancing, task migration and fault tolerance. Virtual
machine provides a framework that determines which tasks are responding and supports naming services so that independently spawned tasks can identify each other and cooperate [122, 123].

Another aspect of virtual machine dynamics relates to efficiency. Computational needs of user applications can change at the time of their program execution. So, a message-passing infrastructure should have a flexible control over the amount of computational power being exploited. For example, consider a typical application, which begins and ends with basically serial computations, but includes several phases of heavy parallel computation. A large Massively Parallel Processor (MPP) system need not be washed out as part of the virtual machine for the serial portions, and can be added just for those portions when it is of most value. Likewise, consider a long-running application in which the user occasionally wishes to attach a graphical front-end to view the computation's progress. Without virtual machine dynamics, the graphical workstation would have to be allocated during the entire computation [122]. MPI does not relate to the dynamics and it is designed to be static in nature to improve performance.

Virtual machine in PVM is responsible for encapsulating and organizing resources for parallel programs. The parallel programmer does not need to manually select every host where tasks are to be executed and then log into each machine in turn to actually spawn the tasks and monitor their execution, the virtual machine provides a simple abstraction to cover the distinct machines. Further, this resource abstraction is carefully layered to allow varying degrees of control. The arbitrary collection of machines then can be treated by the users as uniform computational nodes, in spite of their architectural disparity. Alternatively, the users are free to request for the execution of particular tasks on intended machines with particular data formats, architectures, or even on an explicitly named machine by traversing the increasing levels of detail [123].

Any abstraction for computing resources is not supported by the MPI standards and allows each MPI implementation or user to customize their personal choice of management schemes. This approach of personal choice is good but creates overheads [122].
5.5.11 Fault Tolerance

Fault tolerance is the most important and most critical issue in large scientific computing applications. Without fault tolerance some long running applications cannot be completed ever. When a process is registered to virtual machine or it leaves the virtual machine or the status of the virtual machine changes, it must be notified to the virtual machine [65].

A task can post a notification for any of the tasks from which it expects to receive a message. In this context, the receiving task will get a notify message if any task fails or expires. So getting the notice of related tasks, system can reside in a safe state. A huge loss may happen if a critical task fails just before the completion of an application at last.

In a similar fashion, if a node like an I/O server, critical to an application, fails, the application tasks can post notifies for that node. The tasks will then be informed that the server is replaced with a new one in the virtual machine. When a host exits from a virtual machine by notification to the application tasks, the application tasks adjust with the remaining available resources so that the tasks do not hang [65].

PVM provides more support for fault tolerance and recovery by exposing to the programmer some of the properties of sockets. MPI does less, in the interest of greater portability. Fault tolerance in MPI is an important research topic. MPI-1 does not include any fault tolerance scheme while MPI-2 includes a little of that [65].

5.5.12 Global Name Spaces

A database (name spaces) is created for storing the names of the processes, messages or services with the object of identifying each by its name. Processes can register or unregistered to or from the name space dynamically. Advantage is that processes can be identified independently from the underlying process management and communication environment. The dynamic nature of PVM builds name service extremely useful and convenient. MPI-1 has no functionality that does require name services [122]. The comparison between PVM and MPI is formulated in the table below.
Chapter 5: Review on Parallel Programming Tools

Table 5.2 Comparison between PVM and MPI

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Parameters</th>
<th>MPI-1</th>
<th>MPI-2</th>
<th>PVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Portability Support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Heterogeneity Support</td>
<td>Not</td>
<td>Not</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Interoperability Support</td>
<td>Not</td>
<td>Not</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Virtual Machine Support</td>
<td>Not</td>
<td>Not</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>Topology Support</td>
<td>Yes</td>
<td>Yes</td>
<td>Not</td>
</tr>
<tr>
<td>6</td>
<td>Dynamic Process Group Support</td>
<td>Good</td>
<td>Good</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>Process Control Support</td>
<td>Not Defined</td>
<td>Yes</td>
<td>Fully</td>
</tr>
<tr>
<td>8</td>
<td>Resource Management Support</td>
<td>Not</td>
<td>Not</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>Name Spaces</td>
<td>Not</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

5.6 Summary

We have pointed out mainly the features of two APIs that is MPI and PVM. Now a question arises which API should be used mostly by the programmers? If an application is going to be executed on a single MPP, MPI would be the most suitable option. In this case, the system performance is highly increased. MPI is very rich by the communication functions so it becomes very useful for the application that exploits special communication modes which is absent in PVM. The absence of interoperability and fault tolerance in MPI enhances the communication performance. If an application is going to be executed in heterogeneous platform, PVM is the most suitable option. Since the PVM
is built around the virtual machine concept, the applications can be executed over a collection of platforms of different hosts. PVM contains the functions like dynamic process management, resource management and fault tolerance and interoperability which are the key attributes for heterogeneous computing. The ability to write long running PVM applications that can continue even when hosts or tasks fail or loads change dynamically due to outside influence, is quite important to heterogeneous distributed computing.