A Deep Evaluation of Design Issues and Performances of PVM, MPICH and mpiGAMMA Libraries

MARCO MANIEZZO
ANDREA SANNA
Dipartimento di Informatica ed Automatica
Politecnico di Torino
ITALY
{marco.maniezzo, andrea.sanna}@polito.it

Abstract

There are many ways to program multiprocessor computers and message passing is one of the most popular and widespread approaches. This paper deeply investigates both the design issues and performance of message passing libraries like PVM, MPICH, and mpiGAMMA. We analyze how libraries allow the user to send and receive data packets and how different programming choices can strongly affect the performance of a parallel application. A set of test programs have been used to measure communication delays, latencies due to send and receive procedures, and throughputs.

Keywords : message passing libraries, PVM, MPI, GAMMA project.

1 Introduction

The decision to use message passing to implement parallel code has to be taken after an accurate evaluation of the application to be developed. Using message passing to implement a code is often substantially more work than using compiler directives. For many codes that involve mostly array operations, the native Fortran or C parallel compilers produce as good if not better performing code than a message-passing implementation. On the other hand, there are several situations in which message passing is the best solution. If the problem is "high parallel", i.e., there are serial sections that must be run independently multiple times, or if the code cannot simply be parallelized by making parallel loops, or else when the code has to be highly portable across different parallel architectures, the message passing paradigm should be considered.

The efficiency of a parallel application based on the message passing paradigm is related to the mechanism of sending and receiving of data packets. Network bound programs involve a large number of transmissions and receptions, therefore it is extremely important to clearly identify advantages and drawbacks of the most common technologies known in the literature. A lot of work has been done regarding message passing libraries, but an exhaustive presentation of design issues and performances still is lacking. This paper aims to fill this gap by deeply investigating and analysing three largely widespread message passing libraries: PVM[1], MPICH[2] (an implementation of the MPI[3] standard), and mpiGAMMA[4] (an implementation of MPI based on the Active Message communication mechanism that allows to by-pass the TCP/IP stack in order to minimize latency times and maximize the throughput). This paper aims at helping researchers and developers to identify the best solutions for developing their parallel applications. Communication mechanisms are described in detail and an extensive test section provides experimental results for different *network-bounded* applications. In particular, the behaviour of the libraries are described for blocking and unblocking transmissions as well as for synchronous and asynchronous communications. The impact of data packing/unpacking procedures on communication performances is another issue accurately studied (these procedures can guarantee the independence of the communication also when heterogeneous platforms are involved).

The paper is organized as follows: section 2 reviews PVM, MPI, and mpiGAMMA, section 3 presents the test applications and how they have been used to analyse different communication modes. Section 4 compares the performance results of each parallel communication mode, by a ping pong program. The considerations for the results of the test applications are in section 5; issues for users and developers are in sections 6 and 7.

2 Background

Developing a parallel application consists usually in the transformation of a sequential algorithm into a parallel version; afterwards, the main problem consists in choosing which language and parallel library to use: the most common scientific parallel

applications orient their choice to C language and PVM or MPI parallel standard.

When choosing between PVM and MPI, it is first of all important to consider that PVM is both a standard of definitions and an implementation, while MPI is only a standard of definitions. Hence, it must be also decided which implementation of MPI to consider.

Another important issue to consider is that even if both PVM and MPI are standards of definitions for parallel computation for the message passing paradigm, they have been developed with different conceptual objectives, that we will discuss later in this section; for this reason, the routines are forcibly different, and an useful performance comparison can be done only choosing test applications with usages suitable for both PVM and MPI (also depending on the implementation) library routines.

At the present, MPI is at its second standard definition, MPI-2[5]. We have considered, for our case studies, the first version MPI-1[3] and the ANL MPICH[6] 1.1.2 implementation; the most common parallel scientific applications use MPICH and the optimized version mpiGAMMA is based on MPICH 1.1.2. For PVM, the used version is the last available at the moment: PVM 3.4[7].

2.1 PVM and MPICH conceptual considerations

PVM and MPI have been compared many times, and detailed studies [8][9][10] can be found in the scientific literature; however, what has been mainly in focus, is the different philosophy and objectives of the two standards, rather than a performance evaluation for a particular implementation.

The fact that PVM is a standard and an implementation itself, often leads to confusion, when compared just with MPI standard and not to an implementation of it; what happens is that MPI seems to not provide many important features of PVM, while in truth they are just not mandated by the standard, but are provided by many high-quality implementations.

This is the case of error handling and recovery: in the MPI standard there is not a predefined behavior, other then the listing of specific error indicator values.

It must be observed that there are also differences between PVM and MPI standards, that are not provided in MPI implementations; MPI-1 and MPICH 1.1 do not dynamically manage the parallel processes, while PVM does. This means that a parallel application cannot add or remove processes, during execution. The MPI-2 standard introduces dynamic management of the processes.

PVM provides a set of routines to manage events: a task can be registered in the PVM environment to be "notified" when the status of the virtual machines[1] changes or a task fails; this capability and the dynamic process management makes PVM able to provide *fault tolerance*. Large scale scientific applications can perform long-running simulations, taking days or even weeks, without engendering problems due to the failure of single or multiple tasks; when a failure occurs, a process responsible of managing events can take decisions in order to allow the progression of the simulation.

MPI-1 cannot provide such instruments, since it is meant to be static; MPI-2 provides dynamic processes and a notify scheme, but still has problems being fault-tolerant (for a complete explanation of this concept, we refer to [9]).

The last brief consideration we make on conceptual differences is on *portability*, *heterogeneity* and *interoperability*. A code is *portable* when it is developed on a system and successively works, when compiled on another different system (with a different operative system, for example). Both PVM and MPI standards define procedures in order to be used in portable code.

A system is *heterogeneous* when composed by single different units; for example a cluster composed by PCs with different vendor hardware or, again, with different operative systems. This concept is related to the concept of *interoperability*: a parallel code is *interoperable* when *portable* (and thus can be compiled in an heterogeneous environment) and able to interact with other instances of it, compiled on other different architectures. Both PVM and MPI standards provide instruments to develop code for heterogeneous systems (for example in C or Fortran), but only PVM can claim to be interoperable. The lack of flexibility for MPI, in this contest, is due to the priority of performance in its design.

2.2 PVM and MPI technical considerations

The considerations on the conceptual differences between PVM and MPI permit the programmer to decide the strategy for the management of processes and many other features of the parallel application to develop.

In this subsection we discuss the differences between PVM and MPICH parallel communication procedures, conditioning the choice of data management strategies and the core of parallel computation algorithms.

2.2.1 Communication modes

The possible communication modes, for sending and receiving data are blocking / unblocking, synchronous / asynchronous. It is important to underline the differences between them, because they are often source of confusion.

MPI (and thus MPICH) distinguishes two categories of communication mode: blocking and unblocking. The difference is that blocking communication stops the program execution until the communication buffer is ready to reuse, while the second modality separates communication from computation.

Further distinctions may be moreover introduced within these two categories:

- Synchronous mode: the sender awaits a matching receive from the receiver.
- Ready mode: the sender expects that a matching receive has already been posted, else it returns an error.
- Buffered mode: the sender copies the data to be sent to a buffer it has already allocated for this purpose, and merely awaits the transmission data to be copied.
- Standard mode: the sender does not provide a buffer for the data to be sent, but just forwards the data to the network.

The underlying system provides buffer space to manage data both for transmission and receival. In the blocking mode the sender always waits for the data to be safely transferred either to the transmission buffer in the buffered mode, or to the system in the standard mode and analogously for the other modes. If the size of the data to be sent exceeds the size of the sending buffer (a predefined and system-dependent threshold), the sender stops until all data has been transferred (that is the receiver starts getting data).

The routines used with blocking transmission are in order:

- MPI_Ssend/MPI_Recv
- MPI_Rsend/MPI_Recv
- MPI_Bsend/MPI_Recv
- MPI_Send/MPI_Recv

With unblocking transmission, the sender does not wait for the sending data to be safely transferred to the sending buffer or system. Reusing data before it is correctly transferred can cause errors; for this reason, MPI provides function to test the correct transfer of the data; the unblocking functions are:

- MPI_Issend/MPI_Irecv
- MPI_Irsend/MPI_Irecv
- MPI_Ibsend/MPI_Irecv
- MPI_Isend/MPI_Irecv

while the test functions are:

- MPI_Wait: blocks until completion of send or receive.
- MPI_Probe: blocks until a message arrives.
- MPI_Iprobe: tests a source for incoming messages, without blocking.

PVM does not support such a complete set of routines, but it simply provides an asynchronous transmission model.
The routines available are:

- pvm_psend/pvm_precv
- pvm_send/pvm_recv
- pvm_nrecv/pvm_probe

With the first pair of routines, the sender packs and send data in one step (with three possible criteria, we treat in the next subsection) using a transmission buffer provided by PVM communication layer and does not wait for a matching receive. The receiver stops its flow until data has been received and then directly unpacks it.

The PVM model guarantees the following about message order: if task 1 sends the message A to task 2, and successively sends the message B, the message A will arrive at task 2 before the message B. Moreover, if both messages arrive before task 2 does a receive, then the underlying network management system will ensure that the message A arrives first.

The second pair of functions is used in conjunction with separated packing routines and will be discussed later.

The third group regards unblocking receive (pvm_nrecv), with unblocking procedure (pvm_probe) to test the data arrival.

One of the typical problems that may be found with PVM routines is summarized in the following example:

Process 1	Process 2
pvm_psend(..,size, ..)	pvm_psend(..,size, ..)
pvm_precv()	pvm_precv()

As explained above, PVM manages the data by copying it to a system buffer, both in sending and receiving operations; it is now worth noting that the communication is asynchronous, so the sender can post multiple sends, without waiting any matching receive. The result is that for certain buffer sizes, the two processes will hang, each waiting for the other

to complete the pvm_precv, and this phenomenon is platform and architecture dependent.

As we will explain in section 4, this kind of communication is often desirable, but PVM does not allow the possibility of controlling errors; with MPI the problem is avoidable using buffered sends or non-blocking routines.

2.2.2 Data packing and derived data-types

The procedures discussed in 2.2.1 regard basic transmission, when a single buffer (that is an array of contiguous data) of a basic data type must be transmitted.

When dealing with heterogeneous data types and structures of data, it is desirable to have more general procedures.

In this scenario, PVM offers the packing philosophy: there is a system data buffer (whose dimension is system dependent) in which the sender *packs* data, regardless of data type. The receiver then *unpacks* the data from the received packed buffer with the same order used by the sender.

The routines used are:

- pvm_pkTYPE: where TYPE is a basic data type (int, double, etc.) for packing the data.
- pvm_upkTYPE: for unpacking the data.
- pvm_send/pvm_recv-pvm_nrecv: to send and receive already packed data, with possibility to use non blocking receive.

The system buffer must be initialized with a procedure (pvm_initsend) and a parameter that defines what kind of packing strategy to use:

- PvmDataDefault: the data is coded with XDR[11] translation. This causes a considerable overhead, but permits PVM to manage heterogeneous clusters, since XDR codification is standard.
- PvmDataRaw: the data is copied “as it is”, with no translation, in the system buffer.
- PvmDataInplace: data is not even copied, but directly transferred when the pvm_send procedure is called.

In the latter case the pvm_pk procedures simply record the data buffer pointers, instead of executing memory-to-memory copy of the data. This method achieves the best performances, but the programmer must consider that any modification to the data between the pvm_pk procedures and pvm_send can cause errors.

Packing is the only available strategy for managing complex data structures in PVM and it is optimized in terms of memory-to-memory data copy and in the interaction between packing and sending procedures.

MPI provides a similar strategy, with the MPI_Pack/MPI_Unpack routines. However, unlike in PVM, the user must provide a pre-allocated data buffer for those routines; the buffer size can be calculated at run-time, depending on the size of the data to be sent, with other provided routines.

The packing routines convert all kinds of data into the byte data type, and copy them to the buffer. This buffer can be sent and received with the routines for basic data types, explained in subsection 2.2.1.

This strategy is not optimized and not recommended by MPI implementers, as opposed to MPI *derived data-types* strategy.

In a few words, with the derived data-type strategy the programmer can build a more general data type, containing information on the basic types and sizes of the data to be sent. No memory-to-memory copy is required, but only a phase of *collecting* information (e.g. the physical addresses of data), before using normal transmission routines (see 2.2.1).

Derived data-types provide a powerful instrument to optimize the performance of a parallel application, but require that data structures be designed with particular criteria. Without entering in the deep technical explanation (see [6] for further details), nested structures and not pre-declared structures at compiling time (that is where reading data from a file, for example), can cause problems to be transmitted with derived data types; in this case one must orient to packing routines.

We have analyzed here only point-point communication, which represents the most important issue in parallel communication. Both PVM and MPI provide instruments for broadcast communications, maintaining the base concepts explained in this section; for further information we refer to [1][6].

Following these assumptions, we executed the ping pong test with the basic data types, described in section 4. We did not treat derived data types, since we could not apply them for the porting of Ribbit (see subsection 3.3.1) because of its original organization of the data. Analogously, this strategy was not used in the other MPI application we tested.

2.3 mpiGAMMA

MPICH library resides on top of TCP protocol, which is designed to be general and equipped with instruments for error recovery and safe transmission, nevertheless suffering of memory-to-memory copy problems.

For parallel computation within LAN, is possible to achieve better throughput and minor latency values, implementing the Active Message

protocol[12], for the communication system (level 2 ISO/OSI); GAMMA project[4] is an implementation example.

The aim of GAMMA is to supply a network driver, for Linux clusters, using Active Message protocol, optimized for vendor specific NICs. It is composed by a core of procedures implementing the Active Message algorithm, and an interface made of a set of macros, to the NIC hardware management. In this structure, designed to be general and applicable to any NIC, we developed our implementation[19] of GAMMA optimized for 3COM 3c996 Gigabit Ethernet Card (see section 3.1 for a description of the cluster used for the tests).

The communication mechanisms implemented in the GAMMA driver are made available to application writers through the *GAMMA user library*. The GAMMA library provides support to application launch, process grouping, point-to-point/broadcast communications based on the Active Ports mechanisms, and some collective routines (barrier synchronization, gather, all-gather; more collective routines are being developed).

For a more general use of GAMMA, mpiGAMMA library is available, which is a modified version of MPICH 1.1.2, that interfaces with LAN hardware through the GAMMA driver; the interface with the programmer does not change, and this makes mpiGAMMA easy to use with all MPI applications.

In this paper, we will show the level of improvement achievable with mpiGAMMA, compared to MPICH and PVM implementation of the test applications.

3 Test suites

In this section we describe:

- The environment in which we ran the tests and the characteristics of the machines of the cluster.
- The ping pong test, shortly; details will be discussed in section 4.

The test applications in detail, with the description of the porting to PVM or MPI, which we have realized for some of them.

3.1 The cluster

A cluster of 8 machines AMD K7 1600MHz, with 512MB RAM and a single CPU was used. The machines have a 3COM 3C996 Gigabit Card and are interconnected by a 3COM Gigabit switch. The operative system installed was Red Hat Linux 7.2 with 2.4.20 kernel.

The parallel environments used are PVM 3.4 and MPICH 1.1.2. mpiGAMMA is a modified version of MPICH 1.1.2 realized within the GAMMA project, for which we have specifically developed the optimized C language macros for the interface with the 3COM 3c996 cards.

3.2 Ping Pong Test

The ping pong application is written in C language and implemented in various communication modes, for PVM and MPI standards.

The objective of the program is to measure the time required to send and receive a buffer of data between two processes in a parallel environment, with various communication modes; the process runs on two distinct machines of the cluster.

We measured the full round-trip time with the following algorithm:

```
for(NREP)
{
 get_time(T1)

 #ifdef PACK
 pack(data)
 #endif

 if(MASTER)
 {
  send(data)
  recv(data)
 }
 else
 {
  recv(data)
  send(data)
 }

 #ifdef PACK
 unpack(data)
 #endif

 get_time(T2)

 elapsed_t = T2-T1
}
```

Data is a buffer of elements of C basic data types:

- char 1 byte
- int 4 bytes
- float 4 bytes
- double 8 bytes

The ping pong test is commonly used to measure *throughput* or *latency* of a transmission channel. Throughput indicates how much MB (or Mbit) can be transferred in a second, between two hosts; latency indicates the delay introduced by the network hardware and the computation and transmission of the headers of the packets. To measure the former, the size of the buffer transmitted has to be 10-100MB, while to measure latency, just few bytes are enough.

3.3 The test applications

A fundamental part of our study was centered on the parallel applications we tested. We wanted to benchmark PVM and MPI (with the MPICH and mpiGAMMA implementation), with common use scientific applications, analyzing in detail the performance they obtain with the different parallel environments.

In literature, few parallel applications have been implemented with both PVM and MPI, and even less, if we restrict the selection to MPICH implementation for Linux clusters; it was thus necessary to choose applications that fit our requirements of parallel intensive scientific calculation, using PVM or MPI, and to port them to the other parallel standard.

Not all applications can be ported without modifying the core algorithm, or without using different strategies of data management and transmission for PVM and MPI; we required applications that use similar and comparable communication algorithms for both parallel standards.

Two applications already implemented for both PVM and MPI (POV-Ray 3.1[13] and NetPIPE[14][15]) and two applications that we modified in order to use both environments (Ribbit[16] developed for PVM and ported to MPI; Gromacs 3.2[17] developed for MPI and ported to PVM) were finally chosen. In the next subsections they will be described in detail, with an analysis of the porting work.

3.3.1 Ribbit

Ribbit is a parallel implementation of a rendering algorithm based on the REYES[18] architecture developed by Pixar. The input scene is coded in a text file which describes:

- Geometric primitives: polygons, patches, quadrics and procedural primitives.
- Shaders and functions: the algorithm and the procedures to be used during the rendering phase.
- General scene information: camera position and orientation, image resolutions, number of samples per pixel, shading rate and so on.

The parallel environment used is PVM and the communication strategy adopted involves the packing of data with heterogeneous types and a considerable network utilization; master and slaves accomplish the rendering, interacting in the following phases:

- Initialization: master sends to all the slaves the general information and the algorithm to use for shading.
- Render organization: master communicates to each slave the scene region it must render.
- Data transfer: master sends to the slaves the primitives to be rendered; after the computation is done, slaves send back to master the portion rendered.

The two last points are repeated in a cycle, until every portion of the scene is rendered. It is important to consider that in the data transfer phase, slaves can send back to the master portions of data not completely rendered, according to the REYES algorithm, that will be processed later, increasing communication.

As explained in section 2, MPI standard offers the *derived data type* technology for the organization of data of different data types, in a unique transmission. This improves performance, with regard to pack procedures. The only problem of using derived data type is when the structure of the data is unknown at the compiling time.

Unfortunately, this is precisely the case in Ribbit, where data is read from an input file and allocated dynamically, without restrictions of types and sizes; this is due to the characteristics of PVM, which allows the packing of data, in the transmission buffer, regardless of the total size or number of different data types.

For this reason, the porting to MPI was possible only by maintaining the packing strategy, hence allocating an initial sending buffer (with static definition of its maximum size) and packing into it all the data to be sent.

In section 4 and 5 we discuss the results obtained with the PVM and MPI versions of Ribbit.

3.3.2 POV-Ray 3.1

POV-Ray (Persistence of Vision Ray-Tracer) creates three-dimensional, photo-realistic images using a rendering technique called ray-tracing. It reads in a text file information describing the objects and lighting in a scene and generates

consequently an image of the scene from the camera viewpoint likewise specified in the text file; the scene must be already present in all the machines of the cluster and the parallel rendering consists in a subdivision of the scene in a number of fragments equal to the number of slaves; each slave computes its fragment and then sends its results to the master, which composes the final image; network usage is not exhaustive, yet the application remains useful for benchmark measurements.

The program is available for both PVM and MPICH; we recompiled it for use with mpiGAMMA and measured its performance, as described in section 5.2.

The MPICH version uses data packing, as the PVM version; this leads to worse performances for MPICH as anticipated in section 2 and demonstrated in section 4 and 5.

3.3.3 Gromacs 3.2

Gromacs is an engine that performs molecular dynamics simulations and energy minimization.

These are two of the many techniques that belong to the realm of computational chemistry and molecular modelling. *Computational Chemistry* is a name indicating the use of computational techniques in chemistry, ranging from quantum mechanics of molecules to dynamics of large complex molecular aggregates. *Molecular modelling* indicates the general process of describing complex chemical systems in terms of a realistic atomic models, with the aim of understanding and predicting macroscopic properties based on detailed knowledge on an atomic scale. Often molecular modelling is used to design new materials, for which the accurate prediction of physical properties of realistic systems is required.

Gromacs reads an input file that describes the molecule to analyze and the parameters of the simulation, such as pressure and temperature values and the number of iterations to use to perform the calculation. It then computes the chemical dynamics and produces an output file, which may be used with an utility for the visualization of atom dynamics coming with the Gromacs distribution.

Gromacs is available with MPICH support for parallel computation and the algorithm used is based on the subdivision of the N particles (base elements of the input structure) to the P processors (hosts in the parallel environment, in our case). The processors calculate the dynamics of their set of particles, but this implies an interaction with the particles on the other processors; since they are organised in a ring, the interaction, for every processor, is meant to be with its left and right neighbours. For this reason, the MPI_Sendrecv function is the most suitable, since it accepts two indices (one for sending to, one to receive from). With MPICH, the hosts in the parallel environment are addressed with an index ranging from 0 to P-1; exploiting this feature, the implementation of the Gromacs parallel algorithm, with MPICH becomes easier.

The choice we made for the porting to PVM was to use the pair pvm_psend/pvm_precv, as a direct translation of the MPI_Sendrecv function; in section 2 we discussed the problems that this can create and in the subsection 4.3 we will deeply analyze this kind of parallel issue.

For the other data to be transmitted, Gromacs developers used a data organization strategy which avoids the use of MPI pack procedures (low performances) and MPI derived data types (high coding complexity), thus enabling the use of the normal blocking procedures (MPI_Send/MPI_Recv pairs). This was easily translated in PVM, again with the pvm_psend/pvm_precv pair, following the transmission model described in subsection 4.1.

In the subsection 5.3 we present the results obtained with MPICH, mpiGAMMA and PVM versions of Gromacs, with special attention to the partial times spent specifically in the transmission phases.

3.3.4 NetPIPE

The NetPIPE utility performs simple pingpong tests, bouncing messages of increasing size between two processors. Message sizes are chosen at regular intervals, and also with slight perturbations in order to provide a complete test of the system.

NetPIPE measures the point-to-point communication performance between idle nodes. This provides an upper bound to the performance that an application can achieve, since there is no measurement of the effect that a loaded CPU would have on the communication system.

As mentioned in section 3.3, NetPIPE is available with both PVM and MPICH (we recompiled it for mpiGAMMA library as we did with POV-Ray) and it uses packing of the data for PVM (modifying a header file, it is possible to choose what kind of packing to use: PvmDataDefault, PvmDataRaw, PvmDataInPlace); for MPICH (and thus mpiGAMMA) a standard blocking transmission without packing was used. In section 5.4 we discuss the results obtained by NetPIPE.

4 Ping Pong test results

In this section we analyze the results obtained with the ping pong test application. The applications described in section 3 transfer large amounts of

data, so we were interested to measure throughput with the ping pong test; thus for the communication buffer we chose 60M elements for char (60MB) and 8M for the others data types (32MB for int and float, 64MB for double). We set NREP=10 in order to calculate an average value for the time elapsed.

Once data types and sizes are chosen, it is important to define the kind of send/receive routines to use, in order to compare PVM and MPI. We considered four different sets of transmission modes, as shown in Table 1 and discussed in subsections 4.1, 4.2, 4.3.

PVM		MPI	
Master	Slave	Master	Slave
pvm_psend pvm_precv	pvm_precv pvm_psend	MPI_Send MPI_Recv	MPI_Recv MPI_Send
pvm_pk.. pvm_send pvm_recv pvm_upk..	pvm_recv pvm_upk.. pvm_pk.. pvm_send	MPI_Pack MPI_Send MPI_Recv MPI_Unpack	MPI_Recv MPI_Unpack MPI_Pack MPI_Send
pvm_psend pvm_precv	pvm_psend pvm_precv	MPI_Isend MPI_Irecv MPI_Wait	MPI_Isend MPI_Irecv MPI_Wait
pvm_psend pvm_precv	pvm_psend pvm_precv	MPI_Sendrecv	

Table 1 PVM and MPI routine sets.

4.1 Blocking transmission

A typical master/slave computation is shown in the flow-chart of Figure 1: the master prepares the data and sends it to a slave, who executes the computation and then sends back the results to the master. The routines used for receiving the data must be blocking routines, because both master and slave have to stop the execution until computed data arrives from the corresponding part. As explained in section 2, the receiving routines are blocking for both PVM and MPI, while MPI_Send is blocking and synchronous (we considered the worst case scenario, with data sizes bigger than threshold). Figure 2 shows the flow of the ping pong test for blocking transmission; the most important difference between PVM and MPI is the delay in MPI_Send, which waits for the matching MPI_Recv., while pvm_psend is asynchronous and doesn't wait for any matching receive.

It is clear that the handshake procedure typical of MPI causes a loss of performance if the time needed to transfer the data is comparable to the handshake time.

With large amounts of data, the *receive* phase takes longer time than the handshake and PVM performs the ping pong test with results similar to MPICH.

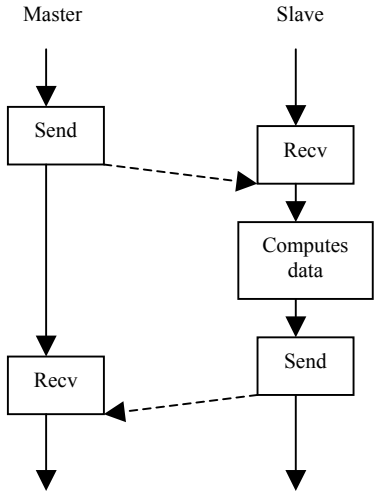

Figure 1 Master/Slave paradigm.

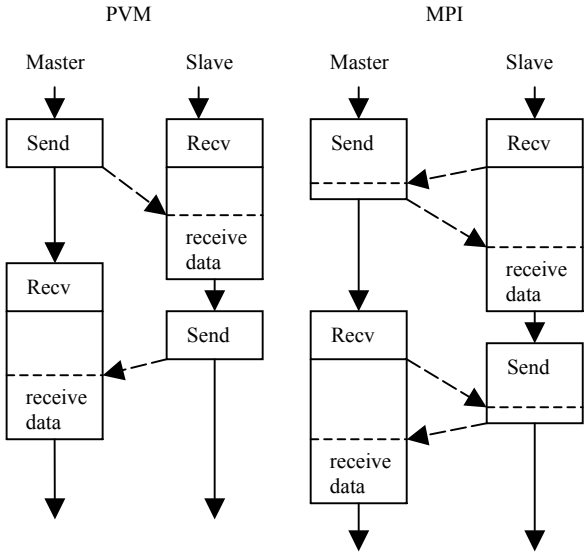

Figure 2 Ping Pong flow for Blocking transmission.

The histogram in Figure 5 displays the measurements we executed with this typology of communication; mpiGAMMA obtains the best results with all data types, as expected. PVM and MPICH differs as follow:

- 32MB transfer: PVM gets best results, performing with 4,3% less total time execution.
- 60-64MB transfer: performances become equal within a 1% difference.

4.2 Blocking transmission with packing

This kind of transmission is often used, as mentioned in section 2, for both PVM and MPI, even though less commonly for MPI, since in this case it is better to use derived data types whenever possible.

For MPI programs that use pack procedures, the transmissions' functioning is as described in section 4.1, plus an overhead due to the packing the data. PVM is more addressed to packing and heterogeneous clusters and pvm_send is meant to be used in junction with pvm_pk routines.

The histogram in Figure 6 demonstrates these theories; PVM without heterogeneous treatment of the data, performs up to 26,4% better then MPICH (with char and PvmDataInPlace pack). If the XDR packing type is used, performance reach worse results for PVM, but they are not really comparable with MPI, since there is no similar treatment of the data for such standard.

mpiGAMMA achieves minor time execution even with packing routines, due to its characteristics of optimization, with up to 17,5% of gain on PVM.

4.3 Unblocking transmission

Some applications can use a different paradigm of communication master/slave, rather than the one explained in section 4.1. Often two processes require the elaboration of separate data and then communicate with each other, the resulting data, as shown in Figure 3.

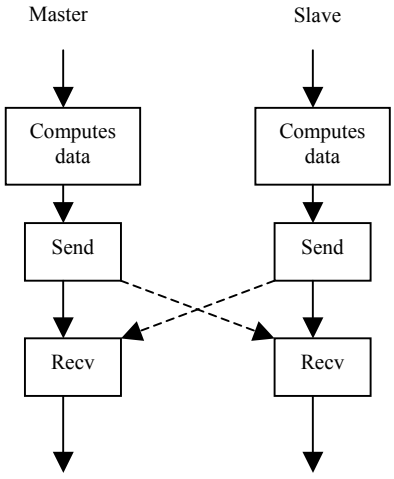

Figure 3 Data exchange.

With this configuration, a synchronous pair send/receive is not desirable; each process does not need to wait until the data sent is actually received, but can continue its own flow.

With MPI, it is possible to implement this algorithm using MPI_Isend/MPI_Wait routines (the third row of Table 1). The wait routine stops the execution until data is received, so in this case the best policy for MPI is the use of the unblocking routines shown in Figure 4.

In this way, the execution flow is stopped only after the call to the receive routine, with the MPI_Waitall call, instead of blocking on the sending routine and on the receiving routine (as happens with MPI_Send and MPI_Recv, with size of the data bigger than the treshold).

For further optimizations, it is possible to use the routine MPI_Sendrecv, which performs in one call, the three routines of Figure 4; there is no improvement of performance, but the code is more compact.

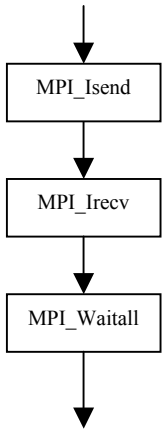

Figure 4 MPI Unblocking set.

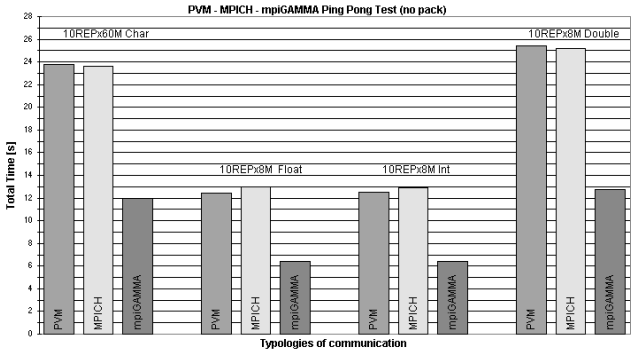

Figure 5 Histogram of Blocking transmission.

With PVM there are no explicit unblocking/blocking routines for sending; pvm_psend (or pvm_send) accomplishes both, with its asynchronous behavior. For receival, it is possible to use the couple pvm_nrecv/pvm_probe,

but it is useful only if other independent computation is possible while waiting for the arrival of the data; in conclusion, for the algorithm in Figure 3, with PVM it is only possible to use the pvm_psend/pvm_precv pair, with the problems described in section 2.

The histogram in Figure 7 collects and compares the results we measured, with asynchronous routines; for MPI we tested MPI_Isend/MPI_Irecv as asynch. 1 and MPI_Sendrecv as asynch. 2: we did not find noticeable total time differences between them. PVM achieves a total execution time 23.3% inferior to MPICH. With buffer size up to 60MB, there were no PVM errors.

mpiGAMMA gets 37,6% best result, on PVM and 52% on MPICH.

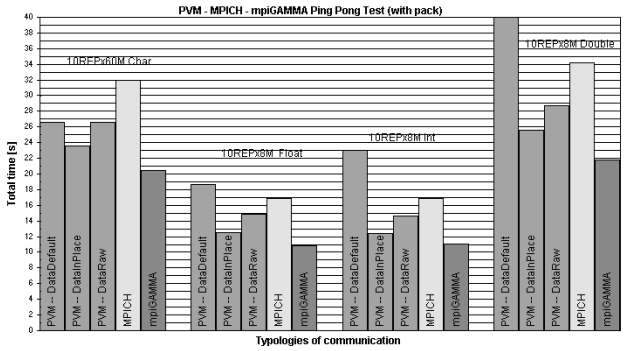

Figure 6 Histogram of blocking transmission with pack.

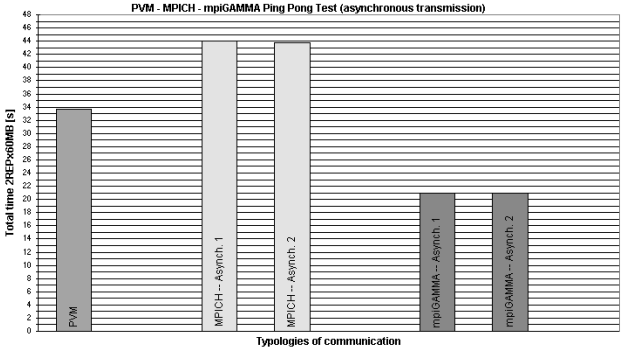

Figure 7 Histogram of asynchronous transmission.

5 Test Applications results

In this section we discuss the results obtained running the applications presented in section 3; as well as compare them with considerations on the various transmission modes related in section 4.

The measurements have been obtained by repeating the execution of each application, 10 times for every configuration of slaves (1, 2 .. up to 7; excluding NetPIPE, which requires only one slave) and by calculating the average time.

5.1 Ribbit results

Ribbit involves a lot of network utilization, especially for the rendering phase where information is continuously transferred forth and back between the master and all the slaves.

The typology of communication is blocking with packing, for both PVM and MPI (for PVM we used PvmDataRaw).

Figure 8 shows time loss percentage trend obtained by increasing the slaves from 1 to 7, for PVM and MPICH, from mpiGAMMA, with two scene resolutions: 800x600 and 1024x768.

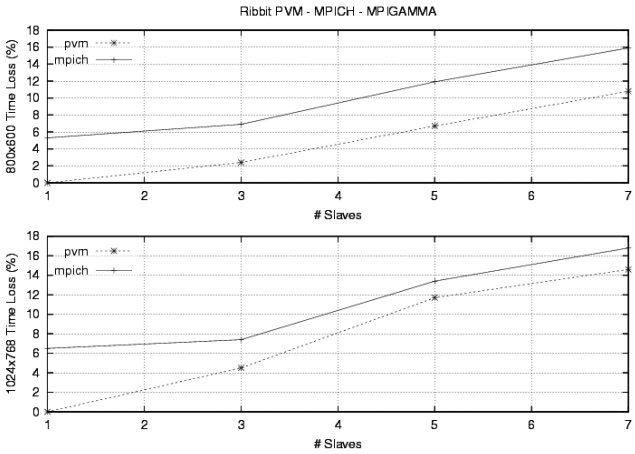

Figure 8 Ribbit output test**.**

As discussed in section 4.2, for the packing routines, PVM obtains better results rather than MPICH; for Ribbit this translates in a gain up to 5%.

Increasing the resolution, the data transfer time rises and becomes more incisive than the packing time. This means less difference between MPICH and PVM (2%) and more time loss from mpiGAMMA (up to 14,6% for PVM and 16.8% from MPICH instead of 10.8% and 16%, respectively), which allows higher throughput.

We chose two resolutions for the scene to render (800x600 and 1024x768), increasing the number of slaves from 1 to 7; with these configurations, the total time execution varies from a minimum of 18.5 seconds up to 97 seconds.

5.2 POV-Ray 3.1 results

POV-Ray is the second rendering software we chose to test; as Ribbit, it uses the blocking/packing typology of communication both for PVM and MPI, but it has a different management of parallel calculation, as explained in section 3. We tested an input scene, with two resolutions: 800x600 and 1024x768, for 1 up to 7 slaves. The graph in Figure 9 presents the time loss percentage: since the scene is already present in all the slaves and there is no

transfer of data, during rendering, the differences of total time execution, do not exceed 0.075% (MPICH maximum time loss, from mpiGAMMA) on a global computation of 122 minutes for the mpiGAMMA version at 1024x768, with 7 slaves. We decided to publish these results the same, because they respect the expected trend: PVM obtains better results than MPICH with a size magnitude compatible with the Ribbit results, increasing the number of slaves; both perform worse rather than mpiGAMMA.

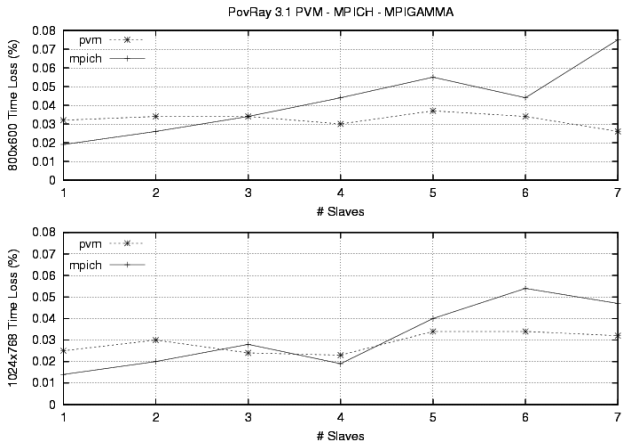

Figure 9 POV-Ray 3.1 output test.

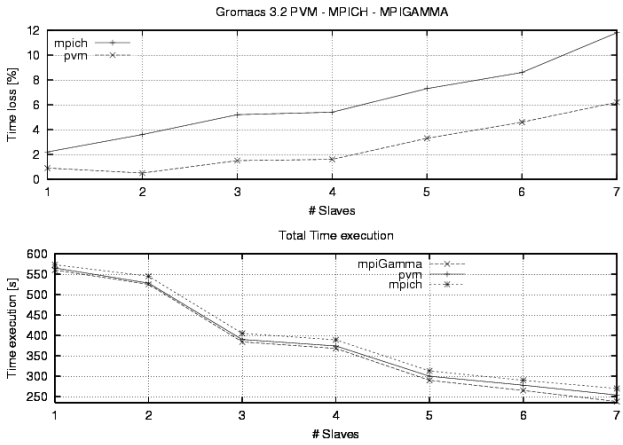

Figure 10 Gromacs 3.2 output test.

5.3 Gromacs 3.2 results

Gromacs differs from Ribbit and POV-Ray, because it does not pack the data exchanged between processes. It also uses the unblocking algorithm described in section 4.3, in addition to normal blocking transmission of section 4.1. As described in section 3, the part of the code that uses the parallel library routines, is separated in a single file and the parallel routines are called in 16 separated procedures; we put measured time for every procedure and then we summed the values relating the total time spent, for every procedure, in the global execution of the program. In Table 2 we show these time measurements, for PVM, MPICH and mpiGAMMA and a program execution with 3 slaves. The total time spent, for all the parallel routines, is in the second column, while the third displays the total time spent for the unblocking communication part; it is immediately evident that this part is the most incisive for the global time, the remaining time spent is not shown. As expected, PVM performs better than MPICH (10% better), due to the unblocking communication used and the major part it takes, loosing only 2.9% from mpiGAMMA.

This leads to the total time percentage losses shown in Figure 10: with 3 slaves PVM gains 4.7% on MPICH and looses 1.5% from mpiGAMMA; this global result, maintains the same proportion of the partial result of the measurements performed on the parallel routines shown in Table 2. Increasing the number of slaves, the gap widens to an almost 12% loss for MPICH and a 6% loss for PVM, from mpiGAMMA, with 7 slaves (for a global computation time of a minimum of 238 seconds, up to 270 seconds); these results confirm the ones shown in histogram of Figure 7, where PVM gets one of its best performance gains on MPICH, among all communication types analyzed.

	Total time all network routines[s]	Total time only unblocking routines[s]
MPICH	209.5	204.5
PVM	187.6	184.4
mpiGAMMA	182.2	178.8

Table 2 Gromacs transmission measurements.

5.4 NetPIPE results

The NetPIPE test has been realized setting the maximum buffer size to 10MB for the transmission, and running the three parallel libraries in exam. In Figure 11 we show the results (throughput in Mbit, varying the packet size) obtained with two different settings, for the PVM version: with no XDR (pvm_send with PvmDataInPlace) and with XDR translation (pvm_send with PvmDataDefault). For both the tests, the MPI version of the program uses MPI_Send/MPI_Recv pairs.

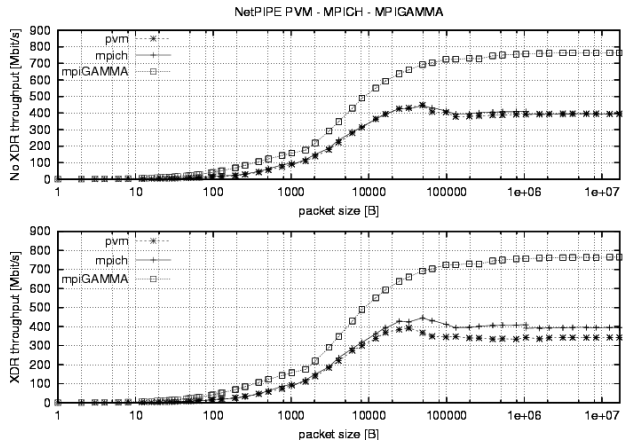

Figure 11 NetPIPE output test.

It is interesting to notice that, with the former test, the results for PVM and MPICH are the same, as can be found comparing the histogram of Figure 5 (MPICH peak) and histogram of Figure 6 (PVM – PvmDataInPlace peak), for float and int (also used in the NetPIPE test).

The second test, with XDR translation, demonstrates that this causes a loss of performance (12.5% on MPICH) for PVM, as described in section 3. mpiGAMMA obtains almost a 100% gain on both PVM and MPICH.

6 User-related issues

Different scenarios that favour MPI rather than PVM and viceversa are presented in order to help developers of parallel applications chose the best solution for their own case.

One of the main limits of the MPI-1 standard, is the absence of dynamic process management; if it is needed, users should choose/favor PVM or MPI-2 standard implementations (MPICH implementation, of the MPI-2 standard, is still under development and available as beta test version). Implementations of MPI-2, such as LAM/MPI[20] are preferred.

On the other hand, PVM provides both a standard and an implementation; this may lead to confusion and deny the user the possibility to establish a standard, for his/her parallel application, and change the implementation used, in the future.

Other important considerations, concerning portability, heterogeneity and interoperability, can be drawn; these features are faced by PVM and MPI in different ways.

When planning the development of a parallel application by MPI, two main different strategies for the management of data should be considered: derived data types and packing. The choice is affected by a better performance of the first rather than user-friendliness of the second. PVM supports only the packing model, however providing better performance in this modality than MPI. Moreover, MPI provides various communication models, while PVM provides only an asynchronous transmission model.

Comparing all the transmission models and parallel routines would be confusing and not useful. The tested applications allowed to investigate three important parallel algorithms:

- Subdivision of the calculation in *n* problems (*n* number of parallel processes), with few interactions between the processes: in this case the network usage is not influent as well as the choice of MPI or PVM.
- Parallel algorithms with high level of interaction between processes: the chosen transmission model deeply influences the performance. Packing data by MPI is not recommended, while the asynchronous model of PVM can lead to errors.
- Deeply nested or complex and dynamic data structures: the derived data types of MPI could be not usable.

Different considerations can be drawn for mpiGAMMA. Its optimization permits to obtain better performance with all the transmission models, compared with both PVM and MPICH. On the other hand, mpiGAMMA is available only for a limited set of NICs and, due to its simplification of network protocols, it is usable only within LANs: parallel computations with global Internet transmissions are not available. For use with an unsupported NIC, the only solution for the programmer is to extend the GAMMA project by coding himself the driver, with the vast difficulties and time that this purpose requires.

Finally, MPI standard lacks the definition of rules for managing fault tolerance. The MPI forum claims fault tolerance is a property of an MPI *program* coupled with an MPI *implementation*. For this reason it is suggested to use implementations such as LAM-based MPI-FT[21] and MPICH-V[22], where fault tolerance is more important than performance (the cost of providing full recovery in all situations is approximately equal to the doubling of communication times). PVM can be considered better for fault tolerance issues, because it provides instruments to manage faults; it is not necessary to find a compromise between performance and fault tolerance that the various MPI implementations provide.

7 Development Issues

Is undoubted that both PVM and MPI are the most widespread libraries and standard for parallel computing. Their projects started over ten years ago and they are still in development and improvement. One of the issues that always roused computer scientists involved in distributed computation are the differences in terms of low level transmission protocols, process management, error handling, routine types.

These concepts lead to many comparisons, as mentioned in section 2, but mainly to one interesting project merging the two standards in one that would contain the best aspects of both PVMPI[23]. The project is dated 1997, when MPI was at its first standard definition and its worst lack was the static management of the processes. PVMPI did not introduce new APIs, but tried to merge the dynamic management of the processes of PVM (by means of PVM routines and subsystems) and performance issues of MPI (again using MPI standard routines implemented by MPICH). Performance was affected by the overhead in the low level transmission protocol due to, essentially, the MPICH layer above the PVM layer. The major difficulty for programmers was to know how to use both PVM and MPI routines.

The project however soon lost interest, when MPI-2 standard was introduced, with its own support for dynamic processes and with further improvement to fault tolerance rather than MPI-1.

There are no other relevant projects in the direction of merging the two standards and thinking of the large number of substantial differences between them, explained in this paper, it can be certainly affirmed that developing a third standard that blends their characteristics and introduces new APIs is not the optimal solution. Neither is it trying to apply features of one to the other thus hoping to optimize the performance: PVM and MPI have their own structure and philosophy to achieve the same objective, by means of different approaches to performance, APIs, error handling and so on.

The right direction for research is to be sought in the individual optimization rather than in the merging of the two approaches. An example is the Ready-Mode Receive function for MPI, developed by Ron Brightwell[24]. In his study he analyzes the ready mode send, of the MPI standard, as we discussed in section 2.2.1. The ready mode send expects that a matching receive has already been posted, thus the matching receive operation is guaranteed that no matching message has yet arrived. For the *ready-mode receive*, there is no need to search a queue of unexpected messages for a possible match (and this is the optimization proposed by Brightwell).

On the other hand, PVM could be improved by working on its asynchronous transmission model discussed in 2.2.1 and its potential run time errors. It also could be enriched by adding more transmission models.

A lot of work could even be done in the GAMMA project; the GAMMA drivers are available only for a limited subset of NICs and they are strongly platform dependent (for instance, there are no available versions for Microsoft OS). Moreover, it supports only MPICH 1.1.2 and substantial improvements could be obtained exploiting PVM and MPI-2.

8 Conclusions

Basing on all the different features of the PVM and MPI libraries described in this article, it is not possible to declare that one solution absolutely better than the other. It is possible to identify a set of issues that can orient users and developers to the choice of a more suitable library for their specific case.

The efforts of future developments for PVM and MPI have been accomplished analyzing and describing main characteristics of both solutions, and testing them by various applications able to show advantages and drawbacks of each choice.

9 Acknowledgements

This project is supported by the Ministero dell'Istruzione dell'Università e della Ricerca (MIUR) in the frame of the COFIN 2001 project: elaborazione ad alte prestazioni per applicazioni con requisiti di elevata intensità computazionale e vincoli di tempo reale.

10 References

[1] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam, "A Users' Guide to PVM Parallel Virtual Machine", Oak Ridge National Laboratory, ORNL/TM-12187, September, 1994.

[2] W. Gropp and E. Lusk and N. Doss and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard", *Parallel Computing*, Vol. 22, No. 6, 1996, pp. 789-828.

[3] Message Passing Interface Forum. "MPI: A message passing interface standard", Technical Report MCS-P342-1193, Computer

Science Department, University of Tennessee, 1994.

[4] G. Ciaccio, G. Chiola, "GAMMA and MPI/GAMMA on Gigabit Ethernet", *Proc. of 7th EuroPVM-MPI, Balatonfured*, Hungary, September 10 - 13, 2000. LNCS 1908, Springer.

[5] http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, ANL University of Tennessee, Knoxville, Tennessee, MPI-2 Extensions to the Message Passing Interface.

[6] http://www-unix.mcs.anl.gov/mpi/mpich/, ANL University of Tennessee, Knoxville, Tennessee, implementation of MPI Message Passing Interface.

[7] http://www.netlib.org/pvm3/index.html, Oak Ridge National Laboratory, Source code for PVM version 3.4.4.

[8] W. Gropp and E. Lusk, "Goals Guiding Design: PVM and MPI", *Proc. of IEEE International Conference on Cluster Computing,* Chicago, Illinois, Sep., 2002, p. 257.

[9] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. "PVM and MPI: A comparison of features". *Calculateurs Paralleles*, Vol. 8, No. 2, 1996, p. 13.

[10] William D. Gropp and Ewing Lusk. "Why are PVM and MPI so different?", *Lecture Notes in Computer Science*, Vol. 1332, 1997, pp 3-10.

[11] SUN Microsystems. "XDR: External Data Representation Standard, RFC 1014", Network Working Group, June 1987.

[12] David Culler, Kim Keeton, Lok Tim Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, and Chad Yoshikawa. "The Generic Active Message Interface Specification", August 1994. Computer Science Division, University of California at Berkeley, White Paper.

[13] POV-Ray: http://www.povray.org/

[14] NetPIPE: http://www.scl.ameslab.gov/netpipe/

[15] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson, "NetPIPE: A Network Protocol Independent Performance Evaluator", ASTED Conference Paper.

[16] O. Lazzarino, A. Sanna, C. Zunino, and F. Lamberti, "A PVM-based parallel implementation of the REYES image rendering architecture", *Proc. of Euro PVM/MPI 2002*, LNCS 2474, 2002, pp.165-173.

[17] GROMACS: http://www.gromacs.org/

[18] R.L. Cook, L. Carpenter and E. Catmull: "The REYES image architecture", *Proc. of Siggraph, ACM Comput*., 21 No. 4 (1987) 95–102.

[19] M. Maniezzo and A.Sanna, "Low latency and high throughput message passing solutions", *Proc. of RTLIA 2003*, pp. 9-12, ISBN: 972-8688-12-1.

[20] Greg Burns and Raja Daoud and James Vaigl: LAM: "An Open Cluster Environment for MPI", *Proc. of Supercomputing Symposium*, pp. 379-386, 1994.

[21] Soulla Louca, Neophytos Neophytou, Arianos Lachanas, and Paraskevas Evrepidou, "MPI-FT: Portable fault tolerance scheme for MPI", *Parallel Processing Letters*, Vol. 10, No. 4, 2000, pp. 371–382.

[22] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cedile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vencent Neri, and Anton Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes", *Proc. of SC 2002. IEEE*, 2002.

[23] G. Fagg, J. Dongarra, and A. Geist, "Heterogeneous MPI Application Interoperation and Process Management under PVMPI", University of Tennessee Computer Science Technical Report, CS-97, June 1997. Submitted to the *Euro PVM-MPI Conference*, Cracow, Poland, Nov., 1997.

[24] Ron Brightwell, "Ready-Mode Receive: An Optimized Receive Function for MPI", *Proc. of 2002 Ninth EuroPVM/MPI Conference*, Sep., 2002.