The Performance of PVM on MPP Systems

Henri Casanova*  Jack Dongarra†  Weicheng Jiang*

August 9, 1995

Abstract

PVM (Parallel Virtual Machine) is a popular standard for writing parallel programs so that they may execute over a network of heterogeneous machines. This paper presents some performance results of PVM on three massively parallel processing systems: the Thinking Machines CM-5, the Intel Paragon, and the IBM SP-2. We describe the basics of the communication model of PVM and its communication routines. We then compare its performance with native message-passing systems on the MPPs.
1 Introduction

PVM (Parallel Virtual Machine) is a software system that allows programmers to use a network of heterogeneous computers, some of which may be massively parallel processing (MPP) systems, as a single multicomputer. In this paper we briefly describe the message-passing features of PVM and discuss PVM’s performance on several MPP systems. See [1] for more details on this software.

All the timings in this report have been obtained with the current version of PVM available at the University of Tennessee. Specifically, we used version 3.3.8 of PVM on the CM-5, the Intel Paragon and on the IBM SP-2. Performance results are summarized in table 2 at the end of this paper and details of the tests are given in appendix A.

2 The Semantics of PVM Message Passing

This section focuses on the message-passing features of PVM.

2.1 Terminology

We define several terms that will be used in this paper to discuss message passing.

**Synchronous Send**: A synchronous send returns only when the receiver has posted a receive.

**Asynchronous Send**: An asynchronous send does not depend on the receiver calling a matching receive.

**Blocking Send**: A blocking send returns as soon as the send buffer is free for reuse, that is, as soon as the last byte of data has been sent or placed in an internal buffer.

**Non-blocking Send**: A non-blocking send returns as soon as possible, that is, as soon as it has posted the send. The buffer might not be free for reuse.

**Blocking Receive**: A blocking receive returns as soon as the data is ready in the receive buffer.

**Non-blocking Receive**: A non-blocking receive returns as soon as possible, that is, either with a flag that the data has not arrived yet or with the data in the receive buffer.
2.2 The Communication Model

The PVM communication model assumes that any task can send a message to any other PVM task. There is no limit to the number of messages and no limit to their size. The communication does not restrict itself to a particular machine's limitations and always assume that sufficient memory is available. The message buffers are allocated dynamically. Therefore, the maximum message size that can be sent or received is limited only by the amount of available memory on a given host. PVM may give the user a cannot get memory error when the sum of incoming messages exceeds the available memory, but PVM doesn’t stop its execution and doesn’t remove the host from the configuration.

According to our terminology, the PVM communication model provides only asynchronous blocking sends. Therefore, the PVM user does not have to worry either about any deadlocks for nonmatching pairs of send-receive or about rewriting into a buffer after it has been sent. PVM provides blocking receives and non-blocking receives.

In PVM3, the option PvmRouteDirect requests that data be transferred directly from task to task by-passing the PVM demon. However this option is ignored on the MPPs, on which PVM is built on the native message passing. The PVM model also guarantees that message order is preserved.

2.3 The Message-Passing Functions in PVM

Until PVM3.3, the only message-passing functions available in PVM were pvm_send() and pvm_recv(). Several additional functions have now been added, as discussed below.

2.3.1 pvm_send(), pvm_recv, pvm_nrecv()

Sending a message requires three steps. First, a PVM buffer must be initialized by a call to pvm_initsend(). Second, the message must be “packed” from the user data space into the PVM buffer by using any combination of the pvm_pck*() routines. PVM takes care of any data encoding and fragmentation. Third, the complete message is sent to another process with pvm_send().

Receiving a message involves two steps. First, the incoming message must be accepted by pvm_recv(), the blocking receive, or by pvm_nrecv(), the non-blocking receive. Second, once the message has arrived, it must be “unpacked” into the user data space with a combination of the pvm_upck*() functions.

During the initialization of the PVM buffer, the user can chose between three different ways of packing the data in this buffer, depending on the parameter passed to pvm_initsend(). The default packing mode is PvmDataDefault. The data is packed from the user space into the PVM-buffer and is encoded according to the XDR format. This mode allows communication over a heterogeneous network (by heterogeneous, we mean a set of computers at least two of which do not have the same data format). A second mode is PvmDataRaw. This mode is similar to the default, but the encoding step is skipped. Thus, PvmDataRaw can be used only between hosts of compatible data formats. It is always more efficient to use PvmDataRaw when running PVM on a single MPP.
In the experiments described in this paper, we used PvmDataDefault once on the CM-5, only to show that it is always highly inefficient to use this data format (see Figure 3(b)). A third option, PvmDataInPlace, leaves the data “in place” in the user data space. During the packing step, PVM simply keeps track of where and how much data is specified. When pvm_send() is called, the data is fetched from the user space and sent over the network (the data is in fact never packed). Using PvmDataInPlace reduces the pack time dramatically and reduces memory requirements. However, care must be taken when using this method as the data should not be modified between the pack call and the send call. Indeed, since PVM keeps only pointers to the data, the data can be modified any time before the send. This situation cannot occur with PvmDataDefault or PvmDataInPlace.

2.3.2 pvm_psend(), pvm_precv()

With PVM 3.3, it is possible to send and receive messages in a single step using pvm_psend() and pvm_precv(). The messages processed by these routines must be exchanged between hosts of compatible data format. Moreover, since there is no packing, the data sent must be contiguous in the sender memory space. In other words, pvm_psend() can be used to send one array of a given data type to one destination, which is a very common type of message in a parallel application. Nevertheless, this feature cannot be used between hosts with incompatible data format, because it involves no data encoding.

2.4 Summary

The Table 1 shows the limitations and possibilities of the different point-to-point systems in PVM3.

2.5 Implementation on MPPs

Figure 1 shows the way the pvm_psend(), pvm_precv(), PvmDataInPlace, and PvmDataRaw are implemented on the MPPs. In the rest of this paper, we will frequently refer to this figure in order to discuss its impact on the performance of PVM. Note that in the figure, we have presented the steps of pvm_psend() and pvm_precv() for two noncontiguous data in the user space. We have also represented the possible extra buffering in the native system on the receiving end. This is the way buffering is done on the Intel Paragon. On the CM-5 and the SP2, however, the buffering is done on the sending end for the native asynchronous blocking send.

3 The CM-5

3.1 The Native Message-Passing System

The CMMD library on the CM-5 enables the user to write message-passing programs. It provides different ways of sending and receiving messages, as we now describe. See [2] for more details.
Figure 1: PVM implementation on MPPs
PVM works on message/passing heterogeneous portability functionality. Motivation routines systems call must be modified to run on a heterogeneous system. Can be used only to exchange one contiguous piece of data of one type. -Direct use of native system -No packing -No unpacking.

Can be used only to exchange one contiguous piece of data of one type. -Direct use of native system -No packing -No unpacking.

Very easy modification to switch to PvmdDataDefault. Can send and receive packed data. -No real packing -Memory saving on the sender.

Very easy modification to switch to PvmdDataDefault. Can send and receive packed data. -Skip the XDR encoding phase of PvmDataRaw.

Can send and receive packed data. -Heterogeneous communication -uses XDR.

Table 1: Summary table of PVM message-passing features

<table>
<thead>
<tr>
<th>PVM message-passing routines</th>
<th>Works on heterogeneous systems</th>
<th>Portability</th>
<th>Functionality</th>
<th>Motivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvm_psend()</td>
<td>No</td>
<td>call must be modified to run on a heterogeneous system</td>
<td>Can be used only to exchange one contiguous piece of data of one type</td>
<td>-Direct use of native system -No packing -No unpacking</td>
</tr>
<tr>
<td>pvm_precv()</td>
<td>No</td>
<td>Very easy modification to switch to PvmdDataDefault</td>
<td>Can send and receive packed data</td>
<td>-Memory saving on the sender</td>
</tr>
<tr>
<td>PvmDataInPlace</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_send()</td>
<td>No</td>
<td>Very easy modification to switch to PvmdDataDefault</td>
<td>Can send and receive packed data</td>
<td>-Skip the XDR encoding phase of PvmDataRaw</td>
</tr>
<tr>
<td>pvm_recv()</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PvmDataRaw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_send()</td>
<td>Yes</td>
<td>Portable</td>
<td>Can send and receive packed data</td>
<td>-Heterogeneous communication -uses XDR</td>
</tr>
<tr>
<td>pvm_recv()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PvmDataDefault</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMD_send_block(), CMD_receive_block() :

According to the terminology defined in 2.1, CMD_send_block() is a synchronous blocking send and CMD_receive_block() is a blocking receive.

CMD_send_async(), CMD_receive_async() :

CMD_send_async() is an asynchronous non-blocking send and CMD_receive_async() is a non-blocking receive. CMMD provides functions to check the completion of the sending and receiving operations.

CMD_send_noblock() :

CMD_send_noblock() is an asynchronous blocking send.

3.2 Comparison of the Native Routines

To assess the native bandwidth and latency, we used CMD_send_block() and CMD_receive_block(). We could also have used CMD_send_async() and CMD_receive_async(), which would have given the same performance. The main advantage of these routines is that they provide the user with the
possibility of overlapping some communications with some computations. It is clear, after some experiments, that `CMMD_send_noblock()` is quite inefficient. This result is surprising for a “ping-pong” test, since normally the receive is always posted and `CMMD_send_noblock()` should be able to send the data without any buffering. Here, on the contrary, it buffers the data systematically, always involving an extra data copy on the sending end. Nevertheless, `CMMD_send_noblock` has several advantages: it cannot lead to a deadlock (as can `CMMD_send_block()`), and the user can reuse its buffer as soon as the call returns (unlike `CMMD_send_async()`). One pitfall in `CMMD_send_noblock` is that it could run out of message descriptors if packets pile up at the sending end.

3.3 The Bandwidth

Figures 2(a) and 2(b) show the bandwidth obtained between two nodes of the CM-5 for

- the native message passing system `CMMD_send_block()`, `CMMD_receive_block()`
- `pvm_psend()`, `pvm_precv()`
- `pvm_send()`, `pvm_recv()` with the PvmDataRaw format
- `pvm_send()`, `pvm_recv()` with the PvmDataInPlace format

The first point to notice in Figure 2(a) is that, as expected, the native message-passing library is the most efficient, with an asymptotic bandwidth of 8.06 Mbytes/sec. Nevertheless, the `pvm_psend()`, `pvm_precv()` bandwidth is fairly close to the performance of the native system. In fact, `pvm_psend()` is built on top of `CMMD_send_async()`, a configuration that explains the good performance (see section 3.1).

We also see that `pvm_send()`, `pvm_recv()` with PvmDataInPlace is much less efficient than `pvm_psend()`, `pvm_precv()`. Two factors explain this inefficiency:

- Unlike `pvm_psend()`, `pvm_precv()`, it involves a real data unpacking on the receiving end (see Figure 1).
- It is built on top of `CMMD_send_noblock()` which we showed much less efficient than `CMMD_send_async()` because of an extra data copy on the sending end (see section 3.2).

Of course, `pvm_send()`, `pvm_recv()` with PvmDataRaw is even less efficient. It is built on top of `CMMD_send_async` but involves an actual data packing-unpacking.

3.4 The Latency

Figure 3(a) shows the transfer time between two nodes for small messages (up to 1024 bytes).

We computed the latencies from Figure 3 using a least squares interpolation. They are given in the following table.
Figure 2: Bandwidth on the CM-5: PVM3 - CMMD

Figure 3: (a) Latency on the CM-5: PVM3 - CMMD - (b) Bandwidth: PvmDataDefault
We observe that the latency for `CMMD_send_block()`–`CMMD_receive_block()` is the lowest. The latency for `pvm_psend()`–`pvm_precv()` is higher, since these routines are built on top of the CMMD routines. Moreover, `pvm_psend()` is much more complex than `CMMD_send_block()`, since it uses `CMMD_send_async()` and accepts incoming messages while waiting for its sending operation to be completed, putting them into a queue (the semantics of `pvm_psend()` implies that no deadlock should occur and that the buffer is ready for reuse when it returns).

The latency of `pvm_send()`–`pvm_recv()` is of course much higher than that of `pvm_psend()`–`pvm_precv()`. This is because of the data packing-unpacking and the use of `CMMD_send_noblock()`. We notice that the latency is higher with `PvmDataInPlace` than with `PvmDataRaw`, which can be seen from Figure 1. With `PvmDataInPlace`, `pvm_send()` has much more “work” to do than with `PvmDataRaw`. Before sending the first data to the receiver, a header must be sent, to inform the receiver about the size of the messages to be expected. This header is built in the PVM space and must be sent separately because it is not contiguous with the data. In the figure, the sending of the header corresponds to the blue arrow number 1. Once it has received the header, the receiver builds a PVM buffer according to the information contained in the header, symbolized by the dashed black arrow on the figure. Then it begins accepting the data in this buffer (blue arrows 2). This process is repeated with the next header if there is one. In our small program, we have only one header of data to transmit. Thus, `pvm_send()` will send a header and then the data. The extra cost of the header is the penalty for short messages.

Note that if we use `PvmDataInPlace` to send `n` noncontiguous different data, `pvm_send()` actually sends `2n` messages. Hence, it is highly inefficient to use `PvmDataInPlace` instead of `PvmDataRaw` to send a large amount of noncontiguous small data.

4 The Intel Paragon

4.1 The Native Message-Passing System

On the Intel Paragon, the NX library enables the user to write message-passing programs. We describe here shortly the different protocols. See [3] for more details.

```c
isend -irecv :
```

`isend` is an asynchronous non-blocking send and `irecv` is a non-blocking receive. NX provides polling functions to check the completion of the send and receive operations.

```c
csend() -crecv() :
```
csend() is an asynchronous blocking send and recv() is a blocking receive.

4.2 Comparison of the Native Routines

This system is similar to CMMD on the CM-5, but it has no synchronous calls (such as CMMD_send_block()). We have seen that on the CM-5, CMMD_send_noblock() is clearly less efficient than the other sending functions. On the Paragon, the performance of csend() is as high as the performance of isend() on a “ping-pong” test. In fact, in this kind of test, the receive is always posted when the data is to be sent. Thus, no extra buffering occurs. The Paragon in this respect is more efficient than the CM-5, which always does an extra copy (see 3.2).

In the following experiments, we used csend()-recv() to assess the native bandwidth and latency.

4.3 The Bandwidth

Figures 4(a) and 4(b) show the bandwidth obtained between two nodes of the Intel Paragon for

- The native message passing system csend()-recv()
- pvm_psend()-pvm_precv()
- pvm_send()-pvm_recv() with the PvmDataRaw format
- pvm_send()-pvm_recv() with the PvmDataInPlace format

As expected, the native message-passing library is the most efficient, with an asymptotic bandwidth of 92 Mbytes/sec. However, the pvm_psend()-pvm_precv() bandwidth is essentially identical for large messages. In fact, pvm_psend() is built on top of isend, a configuration that explains its good performance (see 4.1). In Figure 1, we see that there can be an extra buffering if the message arrives before the receive is posted. The system buffers any incoming message for which no receive has been posted. In a “ping-pong” test, however, the receive is always posted, and this extra buffering never occurs.

As on the CM-5, pvm_send(), pvm_recv() with PvmDataInPlace is much less efficient. First, unlike pvm_psend(), pvm_precv(), it involves an actual data-unpacking on the receiving end, as shown in Figure 1. Second, There may be an extra buffering on the receiving end. We should, however, realize better relative performance than on the CM-5 because csend() does not do a systematic extra data copy, as was the case on the CM-5 with CMMD_send_noblock(). However, there could be an extra data copy as a result of the PvmDataInPlace protocol. When the receiver receives the header, it begins to build a PVM buffer, as explained in section 3.4. Meanwhile, the data may arrive before the receive is posted. In that case, the system does an extra buffering on the receiving end. This is the reason that the relative performance of PvmDataInPlace compared with that of pvm_psend(), pvm_precv() is roughly the same as it is on the CM-5. This also explains the sudden jump when the message size crosses over 10^6 Bytes (1 MB). The default size of the Paragon system buffer is 1 MB, and 3/4 of that is used to buffer incoming messages. The 1-MB
message could not fit in the buffer, so it was held up briefly and then copied into the PVM buffer directly. That, ironically, resulted in better performance.

Of course, \texttt{pvm\_send()} , \texttt{pvm\_recv()} with \texttt{PvmDataRaw} is even less efficient, because it also involves data packing and unpacking. In addition, the message must be buffered by the system, because \texttt{pvm\_recv} polls to check the message length before accepting it.

The little blip in the middle of the \texttt{pvm\_send()}, \texttt{pvm\_recv()} curve corresponds to the system page size.

4.4 The Latency

Figure 5 shows the transfer time between two nodes for small messages (up to 1024 bytes).

We computed the latencies from Figure 5 using a least squares interpolation. They are given in the following table.

<table>
<thead>
<tr>
<th>System</th>
<th>Latency (u-sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>csend</td>
<td>49</td>
</tr>
<tr>
<td>pvm_psend</td>
<td>54</td>
</tr>
<tr>
<td>PvmDataInPlace</td>
<td>332</td>
</tr>
<tr>
<td>PvmDataRaw</td>
<td>320</td>
</tr>
</tbody>
</table>

5 The IBM SP-2

5.1 The Native Message-Passing System

On the IBM SP-2, there are basically three native ways of designing message-passing programs. One can use MPL, a classic message-passing library (see [4]); one can use the private IBM implementation of MPI (Message Passing Interface) (see [6]); or one can use a private implementation of PVM, called PVMe, which corresponds to PVM 3.2.

MPL offers two ways of exchanging messages:

\texttt{mpc\_bsend()}, \texttt{mpc\_brecv()} :

\texttt{mpc\_bsend()} is a synchronous blocking send and \texttt{mpc\_brecv()} is a blocking receive.

\texttt{mpc\_send()}, \texttt{mpc\_recv()} :

\texttt{mpc\_send()} is an asynchronous non-blocking send and \texttt{mpc\_recv()} is a non-blocking receive. MPL provides polling function to check the completion of the send and receive operations.

We used \texttt{mpc\_bsend()}, \texttt{mpc\_brecv()} to do our MPL bandwidth and latency measurements.
Figure 4: Bandwidth on the Intel Paragon: PVM3 - NX

Figure 5: Latency on the Intel Paragon: PVM3, NX
PVMe can be used exactly as PVM 3.2 (that is, without `pvm_psend()` or `pvm_recv()`). It can also be used in two different execution modes:

- **Interrupt**: the CSS (switch) handler signals a task that a message is incoming.
- **No Interrupt**: the CSS handler does not signal an incoming message, and hence may cause deadlock if a large number of messages is exchanged.

The current version of PVM on the SP-2 is built on top of the private implementation of MPI by IBM. We also must note that MPI is implemented not on top of MPL, but at the same level (on top of a common low-level library).

### 5.2 The Bandwidth

In Figures 6(a) and 6(b) we show the bandwidth obtained for

- `mpc_bsend()`, `mpc_brecv()`
- `pvm_psend()`, `pvm_recv()`
- PVMe: PvmDataInPlace Interrupt/No Interrupt
- PVMe: PvmDataRaw Interrupt/No Interrupt
- `pvm_send()`, `pvm_recv()` with PvmDataInPlace
- `pvm_send()`, `pvm_recv()` with PvmDataRaw

In Figure 6(a), the measures for PVMe using PvmDataInPlace are the same regardless of whatever execution mode is used. The measures for PVMe using PvmDataRaw in the “No Interrupt” mode are the same as the measures for PVMe using PvmDataInPlace.

In Figure 6(b), the measures for PVMe using PvmDataRaw are exactly the same as the measures PVMe using PvmDataInPlace.

In Figure 6(a), we see that `pvm_psend`, `pvm_recv` is only slightly less efficient than `mpc_bsend()`, `mpc_brecv()`, which of course the most efficient. As on the CM-5 and the Paragon, PvmDataInPlace is better than PvmDataRaw.

As with the Paragon, the little blip in the middle of the `pvm_send()`, `pvm_recv()` curve corresponds to the system page size.

### 5.3 The Latency

Figure 7 shows the transfer time between two nodes for small messages (up to 1024 bytes). This figure does not show the results for PVMe with PvmDataRaw, since they are the same as the results for PvmDataInPlace.
Figure 6: Bandwidth on the IBM SP-2: PVM3 - PVMe - MPL

Figure 7: Latency on the IBM SP-2: PVM3, PVMe, MPL
We computed the latencies from Figure 7 using a least squares interpolation. They are given in the following table.

<table>
<thead>
<tr>
<th>System</th>
<th>Latency (u-sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpc_bsend</td>
<td>53</td>
</tr>
<tr>
<td>pvm_psend</td>
<td>54</td>
</tr>
<tr>
<td>PvmDataInPlace</td>
<td>224</td>
</tr>
<tr>
<td>PvmDataRaw</td>
<td>202</td>
</tr>
<tr>
<td>PVMe InPlace (Int)</td>
<td>259</td>
</tr>
<tr>
<td>PVMe InPlace (NoInt)</td>
<td>804</td>
</tr>
</tbody>
</table>

The latency of `pvm_psend`, `pvm_precv` is roughly the same as that of `mpc_bsend`, `mpc_brecv`. For the same reason as on the CM-5 and the Paragon, `PvmDataRaw` gives a lower latency than `PvmDataInPlace`.

### 6 Summary

We have assessed the costs of the different PVM routines in terms of memory-to-memory copy and network communication. By network communication, we mean one sending of one message over the network. This message is possibly fragmented by the system. These costs are given for one send-receive operation during our “ping-pong” test, that is with the assumption that the receive is always posted when the send is done. By memory-to-memory copy, we mean the copy of one message from one local buffer to another local buffer. The number of memory-to-memory copies includes the extra system buffering.

The following table gives these costs as illustrated in Figure 1.

<table>
<thead>
<tr>
<th>PVM routines</th>
<th>Memory-to-memory copies</th>
<th>network communications</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pvm_psend()</code></td>
<td>0 on the sender</td>
<td>1 communication</td>
</tr>
<tr>
<td><code>pvm_precv()</code></td>
<td>0 on the receiver</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pvm_send()</code></td>
<td>0 on the sender</td>
<td>2 communications</td>
</tr>
<tr>
<td><code>pvm_recv()</code></td>
<td>2 on the receiver</td>
<td></td>
</tr>
<tr>
<td>PvmDataInPlace</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 copies</td>
<td></td>
</tr>
<tr>
<td><code>pvm_send()</code></td>
<td>1 on the sender</td>
<td>1 communication</td>
</tr>
<tr>
<td><code>pvm_recv()</code></td>
<td>2 on the receiver</td>
<td></td>
</tr>
<tr>
<td>PvmDataRaw</td>
<td>3 copies</td>
<td></td>
</tr>
</tbody>
</table>

The gap between `pvm_send()`, `pvm_recv()` and `pvm_psend()`, `pvm_precv()` seems surprising. To see how much of that can be attributed to the extra data copies, we measure the costs of memory-to-memory copies on all three systems, the results are shown in the following table. The time in the bcopy column is how long it takes to copy the message. The rest is the message roundtrip time
for the three encoding methods, divided by two. The message size is 800 Kbytes, and the time is in microseconds.

<table>
<thead>
<tr>
<th>system</th>
<th>bcopy</th>
<th>psend</th>
<th>send (InPlace)</th>
<th>send (Raw)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2</td>
<td>10528</td>
<td>24168</td>
<td>35720</td>
<td>53326</td>
</tr>
<tr>
<td>CM5</td>
<td>73729</td>
<td>97484</td>
<td>158790</td>
<td>192366</td>
</tr>
<tr>
<td>Paragon</td>
<td>16341</td>
<td>11428</td>
<td>32742</td>
<td>60410</td>
</tr>
</tbody>
</table>

For the CM5, the bcopy time accounted for most of the difference between pvm send(), pvm recv() and pvm_send(), pvm_recv(). For the SP2 and Paragon, it accounted for most of the difference between pvm_send(), pvm_recv() and pvm_send(), pvm_recv() PvmDataInPlace.

7 Conclusion

The philosophy of PVM has always been to keep the user interface simple and to let PVM do the hard work in order to improve the performance. This is why all sends in PVM are asynchronous and blocking. On the other hand, MPP systems usually provide efficient native communication features. PVM’s goal is to use them to improve its performance while keeping its simple message-passing semantic and interface. Therefore, in PVM 3.3, the routines pvm_send() and pvm_recv() have been added. The pvm_send() routine combines the initialize, pack, and send steps into a single call with an orientation toward performance, while pvm_recv() combines the unpacking and the receive steps. The results in this paper clearly show that these new routines yield improved performance and can survive the comparison with the native message-passing systems.

Users who build applications for a homogeneous configuration and have only contiguous data to transmit should benefit from the pvm_send() and pvm_recv() calls. These routines can provide extremely high performance communication, as efficient as the native communication on MPP systems.
<table>
<thead>
<tr>
<th>Protocol</th>
<th>Latency (μ-sec)</th>
<th>Bandwidth (Mbytes/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMMD CMMD_send_block</td>
<td>82</td>
<td>8.25</td>
</tr>
<tr>
<td>PVM pvm_psend</td>
<td>190</td>
<td>8.21</td>
</tr>
<tr>
<td>PVM PvmDataInPlace</td>
<td>858</td>
<td>5.01</td>
</tr>
<tr>
<td>PVM PvmDataRaw</td>
<td>737</td>
<td>4.17</td>
</tr>
</tbody>
</table>

Table 2: Summary table of the performance results
A Details on the tests

**Test program** : We used a very simple program exchanging message of given size between two nodes of any MPP. We then measured an average round-trip time, based on 100 trials. From this average were computed the latency and bandwidth given in this paper.

**CM-5** : We used the CM-5 located at the University of Tennessee. It contains 32 processing nodes. Each of these nodes is a 32 MHz Sparc processor with 32 MBytes of primary memory. The interconnection network is a flat tree, theoretically capable of exchanging data between two nearby nodes at rates up to 20 MBytes/sec.

**Intel Paragon** : We used the Intel Paragon XP/S 5 located at the Oak Ridge National Laboratory. It provides 66 i860 XP compute nodes arranged in a 11 row by 6 column rectangular mesh. Each node has 16MB of memory.

**IBM SP2** : We used the IBM SP2 located at the Cornell Theory Center. All the nodes run at 66.7 MHz The SP2 configuration includes two types of nodes, known as thin nodes and wide nodes. Thin nodes, roughly equivalent to an RS/6000 Model 390, have 128 MBytes memory. Wide nodes have memories that range from 256 MBytes to 2GBytes. The Theory Center’s configuration has 48 wide nodes and 464 thin nodes.
B References

References

PVM 3 User's Guide and Reference Manual,

Thinking Machine Corporation, Cambridge, Massachusetts, May, 1993

[3] Paragon OSF/1 User's Guide,
Intel Supercomputer Systems Division, Beaverton, Oregon, April, 1993

[4] IBM AIX Parallel Environment, Parallel Programming Reference,
IBM, Kingston, New-York, September, 1993

IBM, Kingston, New-York, March, 1995

MPI A Message-Passing Interface Standard,