Intel NX to PVM3.2
Message Passing Conversion Library

Trey Arthur
Computer Sciences Corporation
Hampton, Virginia

and

Michael Nelson
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia

October 1993
Abstract

NASA Langley Research Center has developed a library that allows Intel NX message passing codes to be executed under the more popular and widely supported Parallel Virtual Machine (PVM) message passing library. PVM was developed at Oak Ridge National Labs and has become the de facto standard for message passing. This library will allow the many programs that were developed on the Intel iPSC/860 or Intel Paragon in a Single Program Multiple Data (SPMD) design to be ported to the numerous architectures that PVM (version 3.2) supports. Also, the library adds global operations capability to PVM. A familiarity with Intel NX and PVM message passing is assumed.

1. Introduction

At NASA Langley Research Center (LaRC), the center’s vector supercomputers have become heavily saturated with users’ jobs. Alternatives are being considered to off load some of these jobs to other systems. Among the alternatives considered is the transition of some applications from the vector supercomputers to parallel machines and workstations clusters. With the proliferation of high powered workstations, workstation clustering, in both batch and parallel use, offers an attractive solution to supercomputer saturation.

At NASA LaRC, the Parallel Virtual Machine (PVM) software provides the most popular parallel programming environment. PVM was developed at Oak Ridge National Laboratory...
and has become a defacto standard for message passing (ref. 4). But before PVM had reached this level of popularity, many parallel applications had been developed on the Intel iPSC/860. There was a need to transition these Intel NX message passing (ref. 3) codes to PVM.

This document describes the Intel to PVM, version 3.2 (PVM3) libraries. A familiarity with Intel NX and PVM message passing is assumed. The libraries, \textit{libi2pvm3.a} and \textit{libfi2pvm3.a}, are written in C and contain wrappers for several Intel functions and routines. The executable, \textit{pvmexec}, is a C program which starts the PVM daemons, runs the user application, waits for completion of the slaves, and terminates the PVM daemon processes. If \textit{pvmexec} is run in the Distributed Queing System (DQS) environment (ref. 5), then the PVM daemons will not be started or stopped by \textit{pvmexec}. \textit{pvmexec} is able to detect if it is being run in DQS and will relinquish PVM daemon control to DQS.

The main purpose of the libraries is to allow the user with a code written for a Intel Message Passing Supercomputer in C or FORTRAN to quickly port the code to a workstation cluster using PVM3. To use the libraries in conjunction with the executable \textit{pvmexec} (\textit{pvmexec} is analogous to \textit{cubeezec}$^1$), the user must add two subroutine calls and convert asynchronous message passing calls (e.g., \textit{isend} and \textit{irecv}) to synchronous calls (e.g., \textit{csend} and \textit{crecv}).

Another use of the libraries is to give the PVM3 user access to many of the global libraries which are absent in the standard PVM distribution. To use the global routines without using \textit{pvmexec}, the user should make a call to the \texttt{pvmsetup} routine (see section 4). After the task ids and number of slave processes are known, the \texttt{pvmsetup} routine is called so that the global routines can be used.

### 2. Building the libraries

This library is available via anonymous ftp from

\texttt{blearg.larc.nasa.gov:/pub/pvm/i2pvm3.shar.Z}

Before unpacking \texttt{i2pvm3.shar}, the user should make the directory \texttt{$HOME/pvm3}. To unpack the library, the following should be typed in the user's home directory:

\begin{verbatim}
% sh i2pvm3.shar
\end{verbatim}

\footnote{$^1$\textit{cubeezec} was developed by William J. Nitzberg from the Numerical Aerodynamic Simulation (NAS) Systems Division at NASA Ames Research Center to easily run executable code on the iPSC/860. \textit{cubeezec} is not an Intel supported utility.}
The following should be typed to compile the library:

```
% cd pvm3/i2pvm3
% make
```

This will compile the libraries and pvmexec.c. The libraries are moved to 
$PVM_ROOT/lib/$PVM_ARCH. The executable, pvmexec, is moved to 
$PVM_ROOT/bin/$PVM_ARCH. The include files that the user will need are installed in 
$PVM_ROOT/include.

To compile a program to run in the PVM environment, the following libraries should be 
linked in this order: (libfi2pvm3.a), libi2pvm3.a and libpvm3.a. The following is an example 
compile line for a C and FORTRAN program, respectively:

```
% cc -O -o daria daria.c -L$PVM_ROOT/lib/$PVM_ARCH -li2pvm3 -lpvm3
% f77 -O -o daria.daria.f -L$PVM_ROOT/lib/$PVM_ARCH -lf2pvm3 -li2pvm3 -lpvm3
```

### 3. Use of pvmexec

The libraries can be easily used in conjunction with the executable pvmexec. pvmexec 
starts up the daemon processes, runs the application, waits for the application to finish 
then kills the PVM daemons. If pvmexec is run in the Distributed Queing System (DQS) 
environment (ref. 5), then the PVM daemons will not be started or stopped by pvmexec. 
pvmexec is able to detect if it is being run in DQS and will relinquish PVM daemon control 
to DQS.

When using the library with pvmexec, the first executable line in the code should be a 
call to pvminit(). This routine receives messages from pvmexec. The final call in the user’s 
program should be to pvmquit(). **Failure to call these routines by ALL processes 
will cause pvmexec to hang.** Once the pvmquit() routine has been called by all processes, 
pvmexec will kill the PVM daemons and exit. As noted before, the user must also convert 
asynchronous routines to synchronous routines.

pvmexec recognizes the three options -t, -v, and -V. Option -t is used to specify the 
number of processes to start, -v is verbose mode, and -V prints the version of pvmexec. An 
example for running four processes of the executable node:

```
% pvmexec -v -t 4 node
```

pvmexec will start daemons on all of the hosts in hostfile. hostfile is a PVM host file (ref. 1) 
and is read from the directory in which pvmexec is executed. If hostfile is not present, 
pvmexec will run the all PVM processes on the current workstation.
4. Use of libraries without pvmeexec

The libraries can be used without using pvmeexec, however, it is the user’s responsibility to start and stop the PVM daemons (see ref. 1 for more information). To use the libraries without pvmeexec, make a call to pvmssetup after the task ids and number of slave processes are known. NOTE: if using pvmssetup, do NOT call pvminit or pvmsquit. The following code fragment is an example on how to use pvmssetup.

C example:

```c
mytid = pvm_mytid();
tids[0] = pvm_parent();
if( tids[0] < 0){
    tids[0] = mytid;
pvm_spawn("spmd", (char**)0, 0, ",", NPROC-1, &tids[1]);
pvm_initsend( PvmDataDefault );
pvm_pkint(tids, NPROC, 1);
pvm_mcast(&tids[1], NPROC-1, 0);
}
else {
    pvm_recv(tids[0], 0);
pvm_upkint(tids, NPROC, 1);
}
pvmssetup(tids, NPROC);
```

FORTRAN example:

```fortran
call pvmfmytid( mytid )
call pvmfparent( tids(0) )
if ( tids(0) .lt. 0) then
    tids(0) = mytid
    call pvmspawn( 'spmd', PVMDEFAULT, '*', NPROC-1, tids(1), numt )
call pvmfpack( INTEGER4, tids, NPROC, 1, info )
call pvmscast( NPROC-1, tids(1), 0, info )
else
    call pvmsrecv( tids(0), 0, info )
call pvmsunpack( INTEGER4, tids, NPROC, 1, info )
end if

call pvmssetup( tids, NPROC )
```
## 5. Supported routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sending</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>csend()</td>
<td>csend(msgtype, buf, len, node, pid);</td>
<td>send a message</td>
</tr>
<tr>
<td>csendsi()</td>
<td>csendsi(msgtype, buf, len, node, pid);</td>
<td>send short integer message</td>
</tr>
<tr>
<td>csendi()</td>
<td>csendi(msgtype, buf, len, node, pid);</td>
<td>send an integer message</td>
</tr>
<tr>
<td>csendr()</td>
<td>csendr(msgtype, buf, len, node, pid);</td>
<td>send a real message</td>
</tr>
<tr>
<td>csendd()</td>
<td>csendd(msgtype, buf, len, node, pid);</td>
<td>send a double precision message</td>
</tr>
<tr>
<td><strong>Receiving</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crecv()</td>
<td>crecv(msgtype, buf, len);</td>
<td>receive a message</td>
</tr>
<tr>
<td>crecvs()</td>
<td>crecvs(msgtype, buf, len);</td>
<td>receive short integer message</td>
</tr>
<tr>
<td>crecv()</td>
<td>crecv(msgtype, buf, len);</td>
<td>receive an integer message</td>
</tr>
<tr>
<td>crecvr()</td>
<td>crecvr(msgtype, buf, len);</td>
<td>receive a real message</td>
</tr>
<tr>
<td>crecvd()</td>
<td>crecvd(msgtype, buf, len);</td>
<td>receive a double precision message</td>
</tr>
<tr>
<td><strong>Global</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gdhigh()</td>
<td>gdhigh(buf,num,work);</td>
<td>global double precision MAX</td>
</tr>
<tr>
<td>gdlow()</td>
<td>gdlow(buf,num,work);</td>
<td>global double precision MIN</td>
</tr>
<tr>
<td>gdprod()</td>
<td>gdprod(buf,num,work);</td>
<td>global double precision MULTIPLY</td>
</tr>
<tr>
<td>gdsun()</td>
<td>gdsun(buf,num,work);</td>
<td>global double precision SUM</td>
</tr>
<tr>
<td>ghhigh()</td>
<td>ghhigh(buf,num,work);</td>
<td>global integer MAX</td>
</tr>
<tr>
<td>gilow()</td>
<td>gilow(buf,num,work);</td>
<td>global integer MIN</td>
</tr>
<tr>
<td>giprod()</td>
<td>giprod(buf,num,work);</td>
<td>global integer MULTIPLY</td>
</tr>
<tr>
<td>gisum()</td>
<td>gisum(buf,num,work);</td>
<td>global integer SUM</td>
</tr>
<tr>
<td>gshigh()</td>
<td>gshigh(buf,num,work);</td>
<td>global real MAX</td>
</tr>
<tr>
<td>gslow()</td>
<td>gslow(buf,num,work);</td>
<td>global real MIN</td>
</tr>
<tr>
<td>gsprod()</td>
<td>gsprod(buf,num,work);</td>
<td>global real MULTIPLY</td>
</tr>
<tr>
<td>gssun()</td>
<td>gssun(buf,num,work);</td>
<td>global real SUM</td>
</tr>
<tr>
<td>gsync()</td>
<td>gsync();</td>
<td>synchronization</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvminit()</td>
<td>pvminit();</td>
<td>call when using pvmeexec</td>
</tr>
<tr>
<td>pvmsetup()</td>
<td>pvmsetup(tids,nproc);</td>
<td>call when NOT using pvmeexec</td>
</tr>
<tr>
<td>pvmquit()</td>
<td>pvmquit();</td>
<td>send quit signal to pvmeexec</td>
</tr>
<tr>
<td>mynode()</td>
<td>int mynode();</td>
<td>returns logical process number</td>
</tr>
<tr>
<td>numnodes()</td>
<td>int numnodes();</td>
<td>returns number of processes</td>
</tr>
<tr>
<td>cprobe()</td>
<td>cprobe(msgtype);</td>
<td>wait for a message to arrive</td>
</tr>
<tr>
<td>infocount()</td>
<td>int infocount();</td>
<td>length of message in bytes</td>
</tr>
<tr>
<td>infonode()</td>
<td>int infonode();</td>
<td>node id for sending process</td>
</tr>
<tr>
<td>dclock()</td>
<td>double dclock();</td>
<td>returns wall clock in seconds</td>
</tr>
</tbody>
</table>

Table 1: Supported C routines
<table>
<thead>
<tr>
<th>Routine</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending</td>
<td>csend()</td>
<td>call csend(msgtype, buf, len, node, pid)</td>
</tr>
<tr>
<td>Sending</td>
<td>csendsi()</td>
<td>call csendsi(msgtype, buf, len, node, pid)</td>
</tr>
<tr>
<td>Sending</td>
<td>csendi()</td>
<td>call csendi(msgtype, buf, len, node, pid)</td>
</tr>
<tr>
<td>Sending</td>
<td>csendr()</td>
<td>call csendr(msgtype, buf, len, node, pid)</td>
</tr>
<tr>
<td>Sending</td>
<td>csendd()</td>
<td>call csendd(msgtype, buf, len, node, pid)</td>
</tr>
<tr>
<td>Receiving</td>
<td>crecv()</td>
<td>call crecv(msgtype, buf, len)</td>
</tr>
<tr>
<td>Receiving</td>
<td>crecvsi()</td>
<td>call crecvsi(msgtype, buf, len)</td>
</tr>
<tr>
<td>Receiving</td>
<td>crecvi()</td>
<td>call crecvi(msgtype, buf, len)</td>
</tr>
<tr>
<td>Receiving</td>
<td>crecvr()</td>
<td>call crecvr(msgtype, buf, len)</td>
</tr>
<tr>
<td>Receiving</td>
<td>crecvd()</td>
<td>call crecvd(msgtype, buf, len)</td>
</tr>
<tr>
<td>Global</td>
<td>gdhigh()</td>
<td>call gdhigh(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gdlow()</td>
<td>call gdlow(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gdprod()</td>
<td>call gdprod(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gdsum()</td>
<td>call gdsum(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gihigh()</td>
<td>call gihigh(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gilow()</td>
<td>call gilow(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>giprod()</td>
<td>call giprod(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gisum()</td>
<td>call gisum(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gshigh()</td>
<td>call gshigh(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gslow()</td>
<td>call gslow(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gsprod()</td>
<td>call gsprod(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gssum()</td>
<td>call gssum(buf,num,work)</td>
</tr>
<tr>
<td>Global</td>
<td>gsync()</td>
<td>call gsync()</td>
</tr>
<tr>
<td>Other</td>
<td>pvminit()</td>
<td>call pvminit()</td>
</tr>
<tr>
<td>Other</td>
<td>pvmsetup()</td>
<td>call pvmsetup(tids,nproc)</td>
</tr>
<tr>
<td>Other</td>
<td>pvmquit()</td>
<td>call pvmquit()</td>
</tr>
<tr>
<td>Other</td>
<td>mynode()</td>
<td>integer mynode()</td>
</tr>
<tr>
<td>Other</td>
<td>numnodes()</td>
<td>integer numnodes()</td>
</tr>
<tr>
<td>Other</td>
<td>cprobe()</td>
<td>call cprobe(msgtype)</td>
</tr>
<tr>
<td>Other</td>
<td>infocount()</td>
<td>integer infocount()</td>
</tr>
<tr>
<td>Other</td>
<td>infonode()</td>
<td>integer infonode()</td>
</tr>
<tr>
<td>Other</td>
<td>dclock()</td>
<td>double precision dclock()</td>
</tr>
</tbody>
</table>

Table 2: Supported FORTRAN routines

If the PVM environment has machines with different byte ordering conventions, some additional code changes will be needed. This is because message passing on the Intel is based on sending messages in bytes. If the PVM environment has machines with different byte ordering conventions, the user will need to use a different set of communication routines.
These routines help PVM determine how to send the message. To use these calls, replace \texttt{csend} with \texttt{csendx} where \texttt{x} is either \texttt{si}, \texttt{i}, \texttt{r} or \texttt{d} which stands for short integer, integer, real or double precision, respectively. For example, to send the real variable \(y\) to logical node 2, use this syntax: \texttt{csendr(msgtype, y, 4, 2, 0)}. Note that the message length is still in bytes, so the user only needs to add the appropriate appendix to \texttt{csend}. This message should be received by using the corresponding receive routine: \texttt{crecvr(msgtype, y, 4)}.

6. Unsupported routines

Many NX routines are absent from this library. The supported routines were chosen based on experience in porting from the Intel/i860 to the PVM environment. Many of the asynchronous routines are not supported because it is difficult to emulate these routines in PVM. The easiest solution to this problem is to have the user change asynchronous routines (e.g., \texttt{isend}, \texttt{irecv}) to synchronous communication (e.g., \texttt{csend}, \texttt{crecv}).

7. C Example

Given the following Intel C program:

```
#include <stdio.h>
#include <cube.h>

main()
{
    int iam, nproc;
    float x;

    iam = mynode();
    nproc = numnodes();
    if ( !iam ) {
        x = 20.0;
        csend(100, x, 4, -1, 0);
    } else {
        crecv(100, x, 4);
    }
    gssum(x,1,work);
    if ( !iam ) printf("check: x should equal %d\n",nproc*20.0);
    printf("iam= %d, x= %f\n",iam,x);
}
```
To run this program in a PVM environment using the libi2pvm3.a library, the following code changes would need to be made:

1) change the include file “cube.h” to “nx2pvm.h”
2) change the first executable line to “pvminit();”
3) change the last executable line to “pvmquit();”

Below is the modified C code:

```c
#include <stdio.h>
#include <nx2pvm.h>

main()
{
    int iam, nproc;
    float x, work;

    pvminit();
    iam = mynode();
    nproc = numnodes();
    if ( !iam ) {
        x = 20.0;
        csend(100, x, 4, -1, 0);
    }
    else {
        crecv(100, x, 4);
    }
    gssum(x,1,work);
    if ( !iam ) printf("check: x should equal %d\n",nproc*20.0);
    printf("iam= %d, x= %f\n",iam,x);
    pvmquit();
}
```
The following is a makefile for compiling the program to run on a PVM environment:

```
#
INCLUDEDIR = $(PVM_ROOT)/include
PVMLIB = $(PVM_ROOT)/lib/$(PVM_ARCH)
BDIR = $(PVM_ROOT)/bin
XDIR = $(BDIR)/$(PVM_ARCH)
CLIBS = -li2pv3 -lpvm3

CFLAGS = -g

beavis:
    cc $(CFLAGS) -I$(INCLUDEDIR) -L$(PVMLIB) -o beavis beavis.c $(CLIBS)
    mv beavis $(XDIR)
```

8. FORTRAN Example

Given the following Intel FORTRAN program:

```fortran
program beavis
    include 'fcube.h'

    iam = mynode()
    nproc = numnodes()
    if(iam.eq.0) then
        x = 20.0
        call csend(100, x, 4, -1, 0)
    else
        call crecv(100, x, 4)
    endif
    call gssum(x,1,work)
    if(iam.eq.0) write(*,*) 'check: x should equal ',nproc*20.0
    write(*,*) 'iam = ',iam,' x= ',x
end
```
To run this program in a PVM environment using the libfi2pvm3.a library, the following code changes would need to be made:

1) change the include file “fcube.h” to “fnx2pvm.h”
2) change the first executable line to “call pvminit()”
3) change the last executable line to “call pvmquit()”

Below is the modified FORTRAN code:

```fortran
program beavis
  include 'fnx2pvm.h'
  call pvminit()
  iam = mynode()
  nproc = numnodes()
  if(iam .eq. 0) then
    x = 20.0
    call csend(100, x, 4, -1, 0)
  else
    call crecv(100, x, 4)
  endif
  call gssum(x,1,work)
  if(iam .eq. 0) write(*,*) 'check: x should equal ',nproc*20.0
  write(*,*) 'iam = ',iam, ', x = ',x
  call pvmquit()
end
```

The following is a makefile for compiling the program to run on a PVM environment:

```bash
# PVMLIB = $(PVM_ROOT)/lib/$(PVM_ARCH)
BDIR = $(PVM_ROOT)/bin
XDIR = $(BDIR)/$(PVM_ARCH)
FLIBS = -lfi2pvm3 -li2pvm3 -lpvm3

beavis:
cp $(PVM_ROOT)/include/fnx2pvm.h .
f77 $(FFLAGS) -l$(PVMLIB) -o $O beavis.f $(FLIBS)
mv beavis $(XDIR)
```
9. Executing the examples

The program is compiled and linked by typing make. For compatibility with PVM, the executable beavis is moved to $PVM_ROOT/bin/$PVM_ARCH. To execute beavis over four machines, the file hostfile should be created with each machine name on a separate line (see (ref. 1) for details on how to set up a host file). To execute the code, the following should be typed:

```
% cd $PVM_ROOT/bin/$PVM_ARCH
% pvmexec -v -t 4 beavis
```

Analogous to PVM, all output to the screen is redirected to the file /tmp/pvml.<uid>. To obtain the status of the job while it is running, in another window on any of the machines running PVM, the following should be typed:

```
% pvm
pvm> ps -a
```

10. Summary

This report describes the use of the NASA Langley Research Center library for conversion of Intel NX message passing codes to PVM3.2 message passing codes. If an application is a candidate for conversion, it must be of SPMD design and any asynchronous sends and receives must be converted to synchronous sends and receives. If the intended PVM environment is heterogeneous, some additional code modifications may be necessary.

This library should enable users to quickly port codes developed on the Intel iPSC/860 or Intel Paragon to other environments. This includes workstations clusters or even other parallel computers that provide PVM support. The use of pvmexec emulates the Intel NX environment and should minimize porting difficulties. The use of this library also adds global operations capability to PVM. Additions, modifications, or suggestions are welcome and can be sent to the authors.

References


# Report Title
Intel NX to PVM3.2 Message Passing Conversion Library

## Authors
Trey Arthur
Michael L. Nelson

## Performing Organization Name(s) and Address(es)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

## Sponsor/Monitoring Agency Name(s) and Address(es)
National Aeronautics and Space Administration
Washington, DC 20546-0001

## Subject Terms
Distributed Computing; Parallel Processing; PVM3; Intel NX Message Passing

## Abstract
NASA Langley Research Center has developed a library that allows Intel NX message passing codes to be executed under the more popular and widely supported Parallel Virtual Machine (PVM) message passing library. PVM was developed at Oak Ridge National Labs and has become the de facto standard for message passing. This library will allow the many programs that were developed on the Intel iPSC/860 or Intel Paragon in a Single Program Multiple Data (SPMD) design to be ported to the numerous architectures that PVM (version 3.2) supports. Also, the library adds global operations capability to PVM. A familiarity with Intel NX and PVM message passing is assumed.

## Number of Pages
13

## Price Code
AO3