EASYPVM
EnhAnced Subroutine librarY for PVM

Sami Saarinen

Center for Scientific Computing (CSC),
P.O.Box 405, Tietotie 6, FIN-02101 Espoo, Finland
e-mail: Sami.Saarinen@csc.fi

Abstract. EASYPVM is a message passing library built on top of the Parallel Virtual Machine (PVM) version 3.2. The purpose of EASYPVM is to considerably facilitate the use of PVM-programs by supplying a small set of functions that hide the details of PVM-message passing syntax and process creation. It provides a very coordinated and clear message passing programming approach and eliminates the need of any unnecessary parameters in message passing calls. EASYPVM also introduces some important global commmunication routines and has a very simple interface to PICL/ParaGraph message tracing postprocessor.

1 What is EASYPVM ?

EASYPVM is a message passing library built on top of the Parallel Virtual Machine (PVM) version 3.2. EASYPVM is a simplified message passing interface for those PVM programmers who are aiming to port, test and run MPP-programs on the cluster of workstations, but who are not fully satisfied with the functionality that PVM provides by default.

EASYPVM is in principle available for any platform where PVM is ported. It can be used from C, C++ or Fortran programs. Unlike PVM, EASYPVM-calls are *similar* in any of the aforementioned languages[1]. The performance of EASYPVM is determined by the underlaying PVM-implementation. Also, an EASYPVM-application can still call any of the PVM-routines.

EASYPVM is currently ported for SGI (the development machine), SUN4, IBM RS/6000, Cray, Convex, HP PA-RISC and DEC ALPHA platforms. To obtain a copy of this public domain product, use anonymous ftp (`ftp.funet.fi (128/214.248.6)` under directory `/pub/csc/software/easypvm`. It has been tested with PVM 3.2.6.

2 EASYPVM in more detail

EASYPVM offers the following features that are not available in PVM or would have otherwise required re-inventing the wheel every time a new PVM-program is coded:

[1] Possible name clashes are handled via C-preprocessor.

– A simplified process creation under
  - HOST-NODE programming model
  - NODE-only programming model through generic HOST-program
– Graceful process exits with the careful PVM-daemon shutdowns
– Generic `send()/recv()` -routines that access data in elements rather than bytes
– Global operations including global syncronization, sum, prod, dot product, max/min, concatenation and complete exchange
– `send()/recv()` -operations with 2D-matrix sub-blocks
– Broadcast, Multicast, Store-Forward, SendRecv -functions
– Simple activation of the performance monitoring via PICL/ParaGraph

2.1 Process creation and exit

The underlying logical processor topology in EASYPVM is a 3D-torus found also in the Cray T3D architecture. The so called nearest neighbour -functions and rank indexing of the NODE-processes guarantee flexible access of any NODE-process in a logical *computational cluster*.

EASYPVM is capable for HOST-NODE and NODE-only (SPMD or non-SPMD) approaches. A single HOST can be a user written program or so called `pvmserver`, a generic HOST-program that automatically drives and terminates the NODE-programs. Furthermore, a user written HOST-program can operate simultaneously with several clusters of NODE-programs which need not to be aware of each others.

One weakness in the pure PVM is a process creation. Every time a HOST process creates one or several NODE-processes, nearly all of the following steps must be repeated:

1. Get my PVM task id via `XXXmytid()` -call [2]
2. Get my PVM parent process id from the `XXXparent()` -call
3. Call `XXXspawn(...)` to create the NODE processes
4. Multicast task ids of the NODE processes to every NODE that were just created so that every NODE knows about its siblings

This can be expressed in EASYPVM by a simple subroutine call and providing so called NODE Definition File (`ndffile`) for the HOST process:

```
character*(*) ndffile
CALL createproc(ndffile)
```

Alternatively an SPMD-model can be adopted. In this case the HOST calls `loadproc()` with two parameters: NODE-program executable name and how many copies of the NODE should be initiated.

In the NODE -side the repeated steps are usually:

[2] `XXX` refers to `pvmf` in Fortran or `pvm_` in C/C++ -languages

1. Get my PVM task id via `XXXmytid()` call
2. Get my PVM parent process id from `XXXparent()`
3. Receive task ids of the sibling NODE processes

These can be hidden by the following call:

```
CALL attachproc(me,nproc)
```

This connects the NODE-programs to the PVM-complex and all sibling-related information are made available to them. After the call, the parameters `me` and `nproc` contain the rank of the process (`0..nproc-1`) and how many NODE-processes actually form a computational cluster.

Thus these simple looking function calls hide all the handshaking details from the programmer, who is only responsible for creating the appropriate `ndffile` just before starting his application. An `ndffile` contains typically information about logical 3D-torus dimensions, NODE-executable names and parameter list and on which computers NODEs should be initiated.

For leaving the PVM-complex, EASYPVM offers routine `detachproc()`. This notifies automatically the HOST-process so that the HOST can decrement its counter of the still active NODE-processes. If `detachproc()` was *not* called by the user, then EASYPVM-library takes care of this upon process' exit.

`detachproc()` only leaves the PVM-complex, not exits the process. If program needs to be terminated explicitly, for example, because of some abnormal operation, EASYPVM has another routine available: `killproc(`*procno*`)`. This terminates any process that is a member in a computational cluster; also itself.

Additional features to manage graceful exit and shutdown of any pending PVM-daemons, are implemented via signal handlers in EASYPVM-library and through the system level command `pvmclean`. Both of them guarantee that no processes will remain hanging after abnormal situations. The former one will caught any interrupt signals issued from the keyboard (like `CTRL-C`) or other abnormal situations like segmentation violations. The `pvmclean` does a post-cleaning of any pending processes by killing the PVM-daemons and running down the PVM-complex gracefully.

2.2 Generic message passing calls

With the term *generic message passing* is meant a feature, where a programmer fixes certain data type for subsequent data communication. Data is sent or received using generic `send()` and `recv()` functions and the length of the data arrays is expressed in elements (words) rather than bytes. This offers a nearly trivial port to machines with different word lengths (e.g. Cray vs. other systems) or data representation (e.g. low vs. big-endian).

A typical PVM-application contains several message passing calls that are repeated over and over again, thus making PVM-application hard to read, longsome to code and (thus) errorprone. For example before sending the message, one should do in PVM:

1. initiate a message buffer
2. pack message into that buffer
3. send buffer to a specific destination

This requires (in most cases) three PVM-library calls. In PVM a single message can be formed of different data types. This feature is, however, very seldom used in a real application. For that reason EASYPVM offers a subroutine call `setdatatype()` which fixes the data type for the subsequent message passing operations. A Fortran-code may look like as follows:

```
DOUBLE PRECISION A(N)
... do something with your data A ..
CALL setdatatype(REAL8)    ! Sets data type for subsequent operations
CALL send(dest,100,A,N)    ! Data type is REAL8
... do something else with your data A ..
CALL send(dest,200,A,N)    ! Data type is still REAL8
```

EASYPVM is also capable sending or receiving matrix blocks. Using function calls `send2d()` and `recv2d()` a specific sub-block of a matrix can be accessed. This simplifies programming in many linear algebra applications.

In addition, functions to `broadcast()` or `multicast()` data arrays are available. Also a special function that forwards the just received data buffer before starting to unpack it for itself is available (`storefwd()`). For simple data exchanges function `sendrecv()` can also be used. It simply combines `send()` and `recv()` into one function.

2.3 Global Operations

EASYPVM provides many common global communication routines like global syncronization, summation, product, min-max and concatenate functions. All these functions are also generic, that is, exactly the same function call is suitable for `INTEGER`, `REAL` or `DOUBLE PRECISION` data types, as well as for different `COMPLEX` data types.

If all NODE processes share the same (SPMD-)code, one can syncronize processes in the following manner[3]:

```
CALL gsync()
```

Only after all processes have reached this point in the source code, the NODEs can proceed their execution. This feature is available in a very difficult manner through dynamic process groups in PVM 3.x and was called `barrier()` in older PVM 2.4.x.

Global summation, product, min/max are also available through a single, generic library call (unlike in the Intel's NX-library). It is under user responsibility to assure that the correct data type is in use. A code for `INTEGER` summation may look like as follows:

[3] The routine is exactly the same that is found in Intel's (message passing) NX-library.

```
INTEGER MYVAR, RESULT, mynode
MYVAR = mynode()
CALL setdatatype(INTEGER4)
CALL gsum(MYVAR,1,RESULT)  ! Perform global summation with INTEGERs
```

As result each of the NODE processes have the sum of *local* `MYVAR`'s accumulated in the `RESULT`-variable. It should be noted that the program is essentially the same for the `REAL`-variables, except that function call `setdatatype(INTEGER4)` must be replaced by `setdatatype(REAL4)`.

With the nearest neighbour capability in EASYPVM is meant a set of functions which return the rank index of every nearest neighbour process in a logical 3D-torus. For example, exchanging data with the neighbour processes in left (`west()`) and right (`east()`) sides can be coded as follows:

```
REAL*8 U(0:N+1,0:Mlocal+1)
CALL setdatatype(REAL8)
if (me .lt. nproc-1) then
  CALL send(east(),100,U(1,Mlocal),N)
endif
if (me .gt. 0) then
  CALL recv(west(),100,U(1,0),N)
  CALL send(west(),200,U(1,1),N)
endif
if (me .lt. nproc-1) then
  CALL recv(east(),200,U(1,Mlocal+1),N)
endif
```

2.4 Performance tracing via PICL/ParaGraph

For time being no acceptable way of doing performance tracing is available in PVM. This means difficulties in detection of the communication bottlenecks. Or in tracing which process is sending messages to whom. Or what has been the volume of the communication. EASYPVM removes this hurdle.

By default EASYPVM-library routines are compiled to include performance tracing function calls, that produce PICL[4]-formatted timing records for every communication call issued from a NODE-program. After the run these records are merged and sorted in a timestamp-order for the subsequent post-processing with ParaGraph-tool.

Performance tracing is off by default, but can be activated in a *run-time* by simply defining an environment variable `EASY_PICLTRACE`.

3 Summary and Conclusions

EASYPVM offers a simple way to extend capabilities of the popular PVM. It facilitates testing and porting of the distributed memory programs in a workstation cluster environment written for the "real" MIMD/MPP-systems.

[4] Portable Instrumented Communication Library.

Table 1. A summary of the most important functions in EASYPVM

Function call	Short description
createproc('ndffile')	Creates NODE-processes via ndffile
loadproc('a.out',num)	Starts num 'a.out'-procs
attachproc(me,nproc)	NODE becomes a member of a cluster
mynode()	Returns rank index of my process
myhost()	Returns PVM-task id of my creator
numnodes()	Returns number of NODEs in a cluster
detachproc()	Leave computational cluster
killproc(procno)	Kill specific process in a cluster
send(dest,msgtag,data,nelems)	Sends data of nelems to dest
recv(source,msgtag,data,nelems)	Receives data from source
broadcast(msgtag,data,nelems)	Broadcast data to sibling NODEs
setdatatype(type)	Sets data type for subsequent send()/recv()
getdatatype()	Gets data type currently active
setstride(stride)	Sets stride for subsequent send()/recv()
getstride()	Gets stride currently active
probe(source,msgtag)	Blocking test of received messages
iprobe(source,msgtag)	Non-blocking test of received messages
recvinfo(bufid,source,msgtag,nbytes	Information about received message
gsync()	Global syncronization within a cluster
gsum(input,nelems,output)	Global, elementwise sum of input
gprod(input,nelems,output)	Global, elementwise product of input
gmin(input,nelems,output,where)	Global, elementwise min and its location
gmax(input,nelems,output,where)	Global, elementwise max and its location
gdot(input1,input2,nelems,output)	Global dot product
gcol(data,nelems,output,noutput	Global concatenation of local vectors
gcolx(data,nlengths,output	Global concatenation of local vectors (fast version)
gcxch(data,blksize,datatransp	Complete exchange ("transpose")
east(),west(),north(),south(),up(),down()	Direction functions in a 3D-torus
getprocdim(nx,ny,nz)	Size of the logical 3D process torus
getprocpos(my_x,my_y,my_z)	My position in a 3D-torus

EASYPVM also removes most the PVM-programming burden, but also creates some new barriers related to mixed data type packing that is already available in PVM. However, the nature of EASYPVM is not to restrict PVM-programming in any way, thus all functions available in PVM, are still fully available for EASYPVM applications.

This article was processed using the LTEX macro package with LLNCS style