NIST PowerParallel Primer

Karin A. Remington

Mathematical and Computational Sciences Division, NIST ITL

August 28, 1998

Note: This document was generated automatically from the WWW pages located at URL address http://math.nist.gov/mcsd/Staff/KRemington/Primer/primer.html.

Disclaimer

The mention of specific products, trademarks, or brand names in this Primer is for purposes of identification only. Such mention is not to be interpreted in any way as an endorsement or certification of such products or brands by the National Institute of Standards and Technology. All trademarks mentioned herein belong to their respective owners.

Contents

1 Overview of the NIST SP2 **7**
- 1.1 SP2 hardware information 7
- 1.2 Interactive vs. batch access 8
- 1.3 Sources of information and documentation 8

2 Distributed-memory programming **9**
- 2.1 The "art" of distributed-memory programming 9
- 2.2 SPMD versus MPMD programming 10
- 2.3 Choosing a message passing library: PVM(e) or MPL? 11

3 Message passing with PVM(e) **12**
- 3.1 Setting up the PVM environment 12
- 3.2 Checking the PVM environment 13
- 3.3 Getting started with PVM programming 13
- 3.4 Sending a PVM message 14
- 3.5 Receiving a PVM message 15
- 3.6 Broadcasting a message in PVM 16
- 3.7 Differences between PVM and PVMe 16
- 3.8 PVM/PVMe example in Fortran 16
- 3.9 PVM/PVMe example in C 17

4 Message passing with MPL **18**
- 4.1 Getting started 19
- 4.2 Sending an MPL message 20
- 4.3 Receiving an MPL message 21
- 4.4 Broadcasting a message in MPL 22
- 4.5 MPL example in Fortran 23
- 4.6 MPL example in C 23

5 Compiling message passing programs **24**

6 Using the LoadLeveler batch control system **24**
- 6.1 Batch commands 25
- 6.2 Batch job preparation utilities 25

7 Batch job utilities: Extoll **26**
7.1 Introduction to Extoll 26
7.2 Use of Extoll 26
7.3 Generic Batch Scripts for Message Passing Programs 28
7.4 Sample command file 28

8 Interactive access to the NIST SP2 **30**
8.1 Introduction 30
8.2 Managing message passing programming sessions 31
8.3 Debugging PVM(e) programs 31
8.4 Tutorial examples 32

9 Some tips on debugging parallel programs **33**

10 Converting an MPMD program to SPMD **35**

11 Batch debugging **36**

12 XPVM program monitoring on the SP2 **36**
12.1 Program visualization with XPVM 36
12.2 Requirements 36
12.3 Tutorial example: Matrix transposition 37
12.4 XPVM on the interactive node pool 38

A Interactive introductory README file **40**
README.FIRST 40

B MPL interactive example files **43**
README.debug 43
README.mpl 46
make.mpl 48
mplmanager.f 50
mplworker.f 52

C PVM/PVMe interactive example files **54**
README.debug 54
README.pvm 57
README.pvme 60

make.pvm 64
make.pvme 66
manager.f 68
worker.f 71

D MPL batch example files (C) 73
README.mpl 73
make.mpl 76
mplmanager.c 78
mplworker.c 81
user.cleanup 83
user.setup 84

E MPL batch example files (Fortran) 85
README.mpl 85
make.mpl 88
mplmanager.f 90
mplworker.f 92
user.cleanup 94
user.setup 95

F PVM/PVMe batch example files (C) 96
README.pvm 96
README.pvme 99
make.pvm 102
make.pvme 104
manager.c 106
user.cleanup 109
user.setup 110
worker.c 111

G PVM/PVMe batch example files (Fortran) 113
README.pvm 113
README.pvme 116
make.pvm 119
make.pvme 121

manager.f 123
manager.input.pvm 126
manager.input.pvme 127
user.cleanup 128
user.setup 129
worker.f 130

H SPMD example (PVM/Fortran) 132
spmd.f 132

I XPVM matrix transposition example (PVM/C and Fortran) 137
README.pvm 137
make.pvm 141
realtime.c 143
setup.c 145
transp.f 149
user.cleanup 154
xpvmconnect.c 155

J 157
hostfile.txt 157

Index 158

1 Overview of the NIST SP2

The IBM Scalable POWERparallel System 2 (SP2) is a distributed memory multiprocessor consisting of a collection of RS6000 workstation nodes (running the AIX operating system) together with a Scalable POWERparallel Switch providing communication between the nodes.

1.1 SP2 hardware information

- 30 "SP2" Nodes, with IP addresses grand[1-16].nist.gov and grand[18-31].nist.gov

  Each of these nodes is the rough equivalent of an RS/6000 Model 390.

  - 66.7 MHz clock
  - 64 KB cache line
  - 512 MB of local memory

  NOTE: it has been observed that two of these nodes (grand24 and grand25) have a slightly faster clock than the remaining nodes.

- 6 "Power2 SuperChip" thin nodes, with the IP addresses grand[32-37].nist.gov
  - 120 MHz clock
  - L1 cache: 128KB data/32KB instr
  - 512 MB of local memory
- 1 "SP2-Wide" Node, with the IP address grand17.nist.gov This node is the rough equivalent of an RS/6000 Model 590.
  - 66.7 MHz clock
  - 64 KB cache line
  - 128 MB of local memory
- Approximately 2.5 GB of local disk space in /tmp on each of the nodes.
- A 98B RS/6000 rack system serving as a control workstation, danube.nist.gov.
- Summary table of SP Processor Nodes for comparison¡!figure Summary table of SP Processor Nodes for comparison¿

- SP Switch
  - Two-level cross bar switch
  - TB3 switch adapter with peak hardware performance of 150 MB/s.

For recent changes to the local configuration, see What's New with the SP2 system (http://danube.nist.gov/new.html) at the NIST SP2 website.

1.2 Interactive vs. batch access

The current NIST SP2 configuration allows users to interactively access the front-end machine `danube.nist.gov` and a small test pool of four nodes (grand17-grand20). The remaining nodes are available for batch jobs.

> This interactive pool is intended to be used only when interactivity is *absolutely required*, for example, in program debugging and run-time performance monitoring.

To execute large parallel jobs using the SP2 nodes, users must submit their jobs through a batch system, which partitions the machine into node pools and controls access to these pools so that jobs are single-threaded **(i.e. when you run on a set of SP2 nodes, you are in a "dedicated" environment)**.

1.3 Sources of information and documentation

There is quite a bit of electronic information you can browse through to get acquainted with the SP2 and PVM or MPI.

Locally, you will find IBM-supplied SP2 docs (in postscript) on danube.nist.gov in the directory /doc. For more information on the differences between PVM and MPI, see the section on Choosing a Message Passing Library (p.11).

Online man-pages are another valuable source of information, in particular for quick information about message passing routine syntax. Man-pages are available for both MPI and PVM, and are accessed by typing: `man routine`. Note, although information is provided for Fortran as well as C, *you must request the information using the C routine name.*

Another local source of information is a documentation system called Info-Explorer, located on danube. It has a X-windows user-interface, and so you must remember to set your `$DISPLAY` environment variable to your local console and add danube.nist.gov to your xhost list. Use the command "setenv

DISPLAY your_console.nist.gov:0.0" on danube to set the DISPLAY variable, and the command "xhost +danube.nist.gov" on your console to allow danube to open a window on your console. InfoExplorer can then be invoked with the command "info [-l libraryname]", where *libraryname* can be used to specify the user manual of interest. Currently, the following are available:

```
info -l xlf    (Fortran manuals)
info -l pe     (Programming Environment manuals)
info -l wb     (Workbench manuals)
info -l LoadL  (LoadLeveler manuals)
info -l pvme   (pvm3 manual)
```

If you start up InfoExplorer without a library specified, you can link to one later in the *"Task & Topic Index"* window (not through the *"Welcome to the InfoExplorer..."* window, which also appears) by pulling down the menu labelled "info" and choosing "Library" and "Select".

The SP User Group Home Page (http://spud-web.tc.cornell.edu/HyperNews/get/SPUserGroup.html) at Cornell has links to current discussion groups, educational sites, and a host of other information. To quote their introduction: "This WWW information server is designed for IBM SP users, systems administrators, and other people interested in parallel programming to share information. It is a place to ask questions, seek answers, and contribute knowledge to the other people interested in the IBM SP system."

There is a well-maintained WWW home page for PVM, PVM on Netlib (http://www.epm.ornl.gov/pvm/pvm_home.html) and some very useful SP training materials on the WWW site at the Cornell Theory Center (http://www.tc.cornell.edu/-Edu/Education.and.Training.Materials/Education.html) Note that the CTC was one of the first IBM SP sites and has a long-term affiliation with IBM, which contributes to their excellent support system.

See also the Frequently Asked Questions (FAQ) list for our local site (http://danube.nist.gov/aix_faq.html) .

2 Distributed-memory programming

2.1 The "art" of distributed-memory programming

If you have not already discovered this, you will probably soon realize that there are significant differences in programming a distributed-memory (DM) machine compared to a conventional machine. In fact, some might say there is a real "art" to DM programming, and a way of thinking that just is not required elsewhere.

The primary reason DM machines are more difficult to use is the fact that, not only is the data in memory distributed, but, in general, *the programmer is responsible for ensuring that data is in the right spot at the right time*, typically by using a message passing library to send and receive data across a network to and from processing nodes in the machine. (A notable exception, of course, is virtual shared-memory machines, such as the KSR, which have operating systems designed to manage distributed data without explicit user control.) This responsibility on the shoulders of the user is far from trivial, particularly considering the fact that data movement across a network doesn't always behave predictably. When data messages are delayed due to backlog on the network, for example, program synchronization becomes an issue, and a given program may not behave deterministically - a characteristic that many programmers have always taken for granted and counted on as an indisputable fact.

Where does the "art" come in? Primarily in finding the right way to view an application so that a data distribution which maximizes efficiency comes to the fore. It's likely that with enough effort, virtually any distribution of data across a machine can be made to work. However, if the goal is to have a program that actually runs *faster* on multiple nodes, rather than slower, the most obvious data distribution might not achieve the goal. "Art" might occasionally translate to patience, while the application developer experiments with a range of distributions until the most efficient is found.

2.2 SPMD versus MPMD programming

The SP2 is equipped to run in both Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD) modes. The former requires that each node run an identical (single) program, though the data being processed on each node may differ. The latter allows the user to run different programs on each of the nodes. Probably the most familiar example of MPMD programming is the manager-worker framework, where the user composes two program: a manager program running on one processor to perform I/O or other program set-up functions and then coordinate the efforts of the other processors, and a worker program running on the remaining processors to perform the basic parallel tasks in the application. Often, SPMD programs are designed to mimic this manager-worker model, since there is always a portion of work in an application which it makes sense to perform on only one processor. This mimicking is accomplished by having one of the processors in the pool act alternately as a manager and a worker. *NOTE: One important advantage to adopting the SPMD mimic over the original MPMD manager-worker model is that the resulting code can be run on only one processor, thus greatly facilitating debugging.* We include a conversion to SPMD example (p.10) to illustrate how a typical MPMD manager-worker pair can be converted into a SPMD program.

2.3 Choosing a message passing library: PVM(e) or MPL?

If you have programmed in PVM in the past, and perhaps even have developed your application for workstation clusters or other machines using PVM, you are most likely interested in putting the PVM portability to use by porting your code without modification to the SP2. Unfortunately, *there can be a performance issue regarding your choice of message-passing library.* Standard PVM release 3.3.11 is installed on the SP2, and supports use of the SP2 as both cluster of workstations with ethernet interconnections (RS6K architecture) as a more tightly coupled system using the SP Switch (SP2MPI architecture). To use the switch while maintaining a good portion of PVM portability, IBM has provided their own proprietary version of PVM, called PVMe ("e" for "enhanced").

> **Note:** PVMe *is* able to access the switch, but there are some catches: it is compatible with a previous version of PVM (currently, 3.2.6), and it restricts the user to one PVM process per node - **if you try to spawn more processes than you have nodes, PVMe will fail.**

If you currently use any new features of PVM3.3 (global reduce operations, for example), then you will have to back-track to 3.2.6 compatibility to run on the switch, and, if you are accustomed to being able to overlap PVM processes on a node, you will have some work to do to adjust to the PVMe restriction. On the bright side, IBM seems committed to continuing to upgrade PVMe in the footsteps of PVM, so eventually those nice PVM3.3 features should be added to PVMe.

An alternative is to give up the PVM portability edge and use IBM's own message passing library, MPL. For obvious reasons, IBM has designed this library to use the switch, and it comes complete with global communication routines and other features not found in PVMe. Further, many of IBM's parallel programming tools for tasks such as program visualization and performance monitoring are available only to users of MPL. The structure of the message passing calls is very similar to those of Intel's NX and Thinking Machines' CMMD - so porting is not necessarily a difficult task. If you have familiarity with these other vendors syntax for message passing, using MPL should be pretty straightforward. If your only message passing experience is with PVM, then the sample programs included later in the tutorial may be useful, allowing you to directly compare a PVM(e) program with a corresponding MPL program which performs the same task. Also, later sections Message Passing with PVM(e) (p.12) and Message Passing with MPL (p.18) provide some examples of how to send and receive messages in each library.

3 Message passing with PVM(e)

PVMe is the IBM proprietary version of the widely used PVM message passing library from Oak Ridge National Laboratory. Its compatibility with the public domain PVM package generally lags one release behind. (For example, the current release of PVMe is compatible with PVM 3.2.6)

We will assume that the reader has a basic understanding of the concept of message passing communication of data on distributed memory parallel architectures. PVM(e) is described here primarily by example, but for the interested reader, extensive PVM documentation is available from the PVM authors at PVM on Netlib (http://www.netlib.org/pvm3/index.html) and from the online documentation available on danube (p.8).

3.1 Setting up the PVM environment

Before executing a PVM(e) message passing program, a "Virtual Machine" (VM) must be initiated by the user. This is done by invoking what is known as the PVM(e) daemon, a process which sets up and maintains the information needed for PVM(e) processes to communicate with one another. In general, the user can "customize" the VM by specifying which hosts it should include, the working directory for PVM processes on each host, the path to search for executables on each host, and so on. PVMe behaves slightly differently than PVM, since nodes are controlled through a Resource Manager. Rather than specifying particular nodes for the virtual machine, the user requests a certain number of nodes, and the Resource Manager reserves these nodes for that one particular user. This is to allow the user dedicated access to the High Performance Switch for the duration of their PVMe job.

One difference between PVM and PVMe on the SP2 is how information is supplied to the daemon. For PVM, a command line argument is used to pass the name of a "hostfile". The PVM hostfile is a formatted text file which specifies this information with a simple syntax understood by PVM. An example is found in hostfile (program listing p.157) .

For PVMe, a command line flag to the daemon, "-n ⟨number⟩", specifies the number of nodes to request from the Resource Manager. To override the default settings for the working directory and executable search path, environment variables can be used. See the PVMe description in /doc/pvme/pvme.ps and the section Interactive tutorial examples (p.32) for more information.

3.2 Checking the PVM environment

Both PVM and PVMe provide a utility for checking the status of the VM. This utility, called the PVM console, is started by entering the command "pvm." A variety of query commands can then be entered at the console prompt to determine, for example, which hosts belong to the VM, which processes are running under PVM, etc. See the documentation PVM on Netlib (http://-www.epm.ornl.gov/pvm/pvm_home.html) for detailed information.

3.3 Getting started with PVM programming

With some exceptions (noted in section on differences between PVM and PVMe (p.16)), PVM and PVMe are source code compatible. However, it is *vital* that your program is linked with the appropriate PVM or PVMe message passing libraries; these are NOT interchangeable. A code compiled and linked for PVM must be relinked to run under PVMe. See the section on Compiling message passing programs (p.24) for sample makefiles with the proper library paths for linking PVM/PVMe programs. With this important difference noted, we will use PVM to refer to both PVM and PVMe throughout the remainder of this section,

In coding a message passing program in PVM, calls to communication routines from PVM are included in the same way one makes subroutine calls in Fortran, or function calls in C. A program which requires PVM communication must "enroll" itself in the PVM virtual machine before these calls are made. The call to enroll a program in PVM is:

`call pvmfmytid(nodeid)` or, in C: `nodeid = pvm_mytid();`

3.4 Sending a PVM message

Messages are prepared for sending by packing a buffer with the data items to be included in the message, with one "pack" call for each data item. This allows the user to easily include data of varying types within the same message. A "initialization" call is made just prior to packing to initialize the message buffer. Once packed, the buffer is sent to a specified destination with a "send" call. The initialization call serves primarily to set whether the data is to be converted to the portable XDR format prior to sending (first argument to pvminitsend is PVMDATADEFAULT, or 0), or sent in its raw form (first argument to pvminitsend is PVMDATARAW, or 1). A third possible value for the first argument to the initialization call is PVMDATAINPLACE, which eliminates the copying of the message into the buffer during packing and therefore improves efficiency. Packing in this context copies a pointer to the data and its size into the buffer (note: the data within each pack must occupy contiguous memory locations). *When PVMDATAINPLACE is used, care must be taken not to modify the data before the message is actually received.* For details about the trade-offs between these initialization schemes, the reader is referred to the PVM or PVMe reference guides. The examples provided in this primer use the PVMDATADEFAULT option, which give the highest degree of portability and protection from unintentional data overwrites.

Sample of Fortran message sending with PVM:

```
c  Initialize the buffer bufid:
      call pvmfinitsend(0, bufid)
c  Pack the real value rstart:
```

```
      call pvmfpack(REAL8, rstart, 1, 1, rc)
c  Pack the real 10-element vector data:
      call pvmfpack(REAL8, data, 10, 1, rc)
c  Pack the integer*4 value chunksize:
      call pvmfpack(INTEGER4, chunksize, 1, 1, rc)
c  Send the message to the node given in tids(1) with a 'tag' of 100:
      call pvmfsend(tids(1), 100, rc)
```

Sample of C message sending with PVM:

```
/* Initialize the buffer bufid (to use default XDR encoding): */
      bufid = pvm_initsend(PvmDataDefault)
/* Pack the real value rstart: */
      rc = pvm_pkdouble(&rstart, 1, 1)
/* Pack the real 10-element vector data: */
      rc = pvm_pkdouble(data, 10, 1)
/* Pack the integer*4 value chunksize: */
      rc = pvm_pkint(&chunksize, 1, 1)
/* Send the message to the node given in tids[0] with 'tag' 100:*/
      rc = pvm_send(tids[0], 100)
```

3.5 Receiving a PVM message

Messages can be received selectively based on a combination of the source (which node is sending the message) and tag (a qualifier specified by the user in the corresponding "send" call). A "wildcard" value (-1) can be specified for one or both of these receive call arguments to allow a message to be received regardless of the source or tag. When a message fitting the specified qualifications arrives at the node, it is received into a buffer. To retrieve the message contents, the buffer must be unpacked, *in the same order in which it was packed at the sending end.*

Sample of Fortran message receiving with PVM:

```
c  Receive a message from node hostid with tag 100 into a buffer:
      call pvmfrecv(hostid, 100, bufid)
c  Unpack the real value rstart:
      call pvmfunpack(REAL8, rstart, 1, 1, rc)
c  Unpack the real 10-element vector data:
      call pvmfunpack(REAL8, data, 10, 1, rc)
c  Unpack the integer*4 value chunksize:
      call pvmfunpack(INTEGER4, chunksize, 1, 1, rc)
```

Sample of C message receiving with PVM:

```
/* Receive a message from node hostid with tag 100 into a buffer: */
      pvm_recv(hostid, 100)
/* Unpack the real value rstart: */
      rc = pvm_upkdouble(&rstart, 1, 1)
/* Unpack the real 10-element vector data: */
      rc = pvm_upkdouble(data, 10, 1)
/* Unpack the integer*4 value chunksize: */
      rc = pvm_upkint(&chunksize, 1, 1)
```

3.6 Broadcasting a message in PVM

Work in progress...

3.7 Differences between PVM and PVMe

Work in progress...

- Number of processes per node limited in PVMe
- Only one daemon process for PVMe. For PVM, in contrast, there is a main daemon on the parent node, and a sub-daemon on each node in the VM.
- Availability of pvm_catchout, and routing of node stdout
- The PVMe version of pvm_config() has a different (longer) argument list than public domain PVM. Read PVMe man pages for correct call; This extra parameter will cause an error if compiled with the public domain PVM.
- Generation of trace information

3.8 PVM/PVMe example in Fortran

On danube.nist.gov, the directory /usr/local/doc/Examples/pvm+pvme/Fortran contains the following simple Fortran example for PVM(e):

```
Filename        Style   Description
--------        -----   -----------
manager.f       MPMD    A basic "hello world" type program
worker.f
```

This directory includes the following files:

- README.pvm (program listing p.113)
- README.pvme (program listing p.116)
- make.pvm (program listing p.119)
- make.pvme (program listing p.121)
- manager.f (program listing p.123)
- manager.input.pvm (program listing p.126)
- manager.input.pvme (program listing p.127)
- user.cleanup (program listing p.128)
- user.setup (program listing p.129)
- worker.f (program listing p.130)

To run the sample program in batch under ** ORNL PVM **, follow the instructions given in README.pvm (program listing p.113)

To run the sample program in batch under ** IBM PVMe **, follow the instructions given in README.pvme (program listing p.116)

3.9 PVM/PVMe example in C

On danube.nist.gov, the directory /usr/local/doc/Examples/pvm+pvme/C contains the following simple C example for PVM(e):

```
Filename        Style   Description
--------        -----   -----------
manager.c       MPMD    A basic "hello world" type program
worker.c
```

This directory includes the following files:

- README.pvm (program listing p.96)
- README.pvme (program listing p.99)
- make.pvm (program listing p.102)
- make.pvme (program listing p.104)
- manager.c (program listing p.106)
- user.cleanup (program listing p.109)
- user.setup (program listing p.110)
- worker.c (program listing p.111)

To run the sample program in batch under ** ORNL PVM **, follow the instructions given in README.pvm (program listing p.96)

To run the sample program in batch under ** IBM PVMe **, follow the instructions given in README.pvme (program listing p.99)

4 Message passing with MPL

MPL is the IBM proprietary message passing library. It doesn't feature the same portability that PVM boasts, however, a number of tools have been developed specifically for the SP2 which require the use of MPL message passing. (Information on these tools will soon be added to this primer.)

We will assume that the reader has a basic understanding of the concept of message passing communication of data on distributed-memory parallel architectures. MPL is described here primarily by example, but for the interested reader, MPL documentation (p.8) is available online.

The function/subroutine calls covered in this tutorial are:

```
In C:                In Fortran:
----------------------------------------
mpc_environ()        call MP_ENVIRON()
mpc_task_query()     call MP_TASK_QUERY()
mpc_send()           call MP_SEND()
```

```
mpc_wait()            call MP_WAIT()
mpc_recv()            call MP_RECV()
mpc_brecv()           call MP_BRECV()
mpc_bcast()           call MP_BCAST()
```

4.1 Getting started

As with PVM, in MPL one includes calls to communication routines in the same way one makes subroutine calls in Fortran, or function calls in C. Corresponding to the pvm_mytid() call in PVM, MPL provides the MP_ENVIRON() call to determine the number of processors (set at runtime) and the task id of the processor running a particular instance of the program. The syntax of the MP_ENVIRON call is:

```
call MP_ENVIRON(nprocs,nodeid)
```

Or, in C:

```
rc=mpc_environ(&nprocs, &nodeid);
```

MPL provides the MP_TASK_QUERY call to obtain information about system variables and constants (beyond what's returned by the MP_ENVIRON call). Depending on the last argument to the MP_TASK_QUERY call (the "query-type"), various information can be retrieved (see the man-page for complete information). The most common use of this call is to obtain the system values for "dontcare" and "allgrp", which are the "wildcard" value, used for indiscriminately receiving messages, and the default processor group containing all nodes, often used in broadcasting information across the machine.

Sample of a Fortran MP_TASK_QUERY call:

```
      integer nbuf(4), dontcare, allgrp
c
c     Learn value of allgrp and dontcare, for later use with broadcast:
c
      call MP_TASK_QUERY(nbuf, 4, 3)
      dontcare = nbuf(1)
      allgrp = nbuf(4)
```

Sample of a C mpc_task_query() call:

```
   int nbuf[4], allgrp, dontcare;
/*
/  Learn the value of allgrp and dontcare, for use with broadcast.
*/
   rc = mpc_task_query(nbuf, 4, 3);
   dontcare = nbuf[0];
   allgrp = nbuf[3];
```

4.2 Sending an MPL message

Messages are sent in MPL with a call to MP_SEND (or mpc_send) for each piece of data to be sent (contrasting with the pack and send model in PVM). The call to the send routine returns immediately, without waiting for the receive to actually take place at the destination node. Because of this "asynchronous" behavior, it is important to pair a send call with a call to MP_WAIT (or mpc_wait) if execution should not proceed until the destination processor has actually received the data. For example, if the data is going to be modified at the source after a send, a wait call before this modification will ensure that the original data is transferred, rather than the modified data.

Sample of Fortran message sending with MPL:

```
c
c       Send msgleng consecutive bytes of data, starting with
c       location nodemsg, to node 0 (host) with a tag of 'msgid'
c
        call MP_SEND(nodemsg, msgleng, 0, 100, msgid)
c
c       Before proceeding, wait for nodemsg to be received at the host:
c
        call MP_WAIT(msgid, nbytes)
c
c       Now, nodemsg can be safely modified...
c
```

Sample of C message sending with MPL:

```
/*
```

```
/     Send msgleng consecutive bytes of data, starting with
/     location nodemsg, to node 0 (host) with a tag of 'msgid'
*/
      rc = mpc_send(nodemsg, msgleng, 0, 100, msgid);
/*
/     Before proceeding, wait for nodemsg to be received at the host:
*/
      rc = mpc_wait(&msgid, &nbytes)
      if (rc) {
         print("Error in mpc_wait(). Return value: %i\nExiting.\n", rc);
         exit(-1);
      }
/*
/     Now, nodemsg can be safely modified...
*/
```

4.3 Receiving an MPL message

Messages can be received selectively based on a combination of the source (which node is sending the message) and tag (a qualifier specified by the user in the corresponding "send" call). A "wildcard" value can be specified for one or both of these receive call arguments to allow a message to be received regardless of the source or tag. When a message fitting the specified qualifications arrives at the node, it is copied into the location specified in the receive call. Unless the specific "blocking" receive call is made (MP_BRECV), the program will not wait for the message to actually arrive before it continues processing. Therefore, if you need to use the data being received, you must either use the MP_BRECV call, or post a MP_WAIT call which will halt execution until the message is received. It is the non-blocking strategy which allows for the possibility of using cycles during network delays to perform useful work. However, it also can give the false impression that data has been received when it has not, thus creating difficult to locate bugs!

Sample of Fortran message receiving with MPL:

```
c
c  Receive a message msgleng bytes long from node i with tag 100 into the
c  consecutive memory locations beginning at 'nodemsg'
c  NOTE: This is a 'blocking' receive.
c
```

```
        call MP_BRECV(nodemsg, msgleng, i, 100, msgid)
```

Sample of C message receiving with MPL:

```
/*
/  Receive a message msgleng bytes long from node i with tag 100 into the
/  consecutive memory locations beginning at 'nodemsg'
/  NOTE: This is a 'blocking' receive.
*/
      rc = mpc_brecv(nodemsg, msgleng, &i, &type, &msgid);
```

4.4 Broadcasting a message in MPL

Broadcasts are an easy way to distribute data from one node to all other nodes (for example, sending program parameters from a host program to the node programs). The MP_BCAST command performs the broadcast, taking the location of the data item to be sent as the first argument, the length of the message as the second argument, and the source node and participants as the final two arguments. Typically, all nodes are included in the participant list, and there is a special value used to indicate this. The broadcast can be tailored to use only subsets of the nodes, however. See the online POE documentation for more details.

Sample of a Fortran broadcast with MPL:

```
c
c     Broadcast the 8 byte double rstart from node 0 to all others:
c
      call MP_BCAST(rstart,8,0,allgrp)
c
c     Broadcast the 4 byte integer chunksize from node 0 to all others:
c
      call MP_BCAST(chunksize,4,0,allgrp)
```

Sample of a C broadcast with MPL:

```
/* Broadcast the 8 byte double rstart from node 0 to all others:      */
   rc = mpc_bcast(&rstart,sizeof(double),nodeid,allgrp);
/* Broadcast the 4 byte integer chunksize from node 0 to all others: */
   rc = mpc_bcast(&chunksize,sizeof(int),nodeid,allgrp);
```

4.5 MPL example in Fortran

On danube.nist.gov, the directory /usr/local/doc/Examples/mpl/Fortran contains the following simple Fortran example for MPL:

```
Filename          Style   Description
--------          -----   -----------
mplmanager.f      MPMD    A basic "hello world" type program
mplworker.f
```

This directory includes the following files:

- README.mpl (program listing p.85)
- make.mpl (program listing p.88)
- mplmanager.f (program listing p.90)
- mplworker.f (program listing p.92)
- user.cleanup (program listing p.94)
- user.setup (program listing p.95)

To run the sample program in batch under ** IBM MPL **, follow the instructions given in README.mpl (program listing p.85)

4.6 MPL example in C

On danube.nist.gov, the directory /usr/local/doc/Examples/mpl/C contains the following simple C example for MPL:

```
Filename          Style   Description
--------          -----   -----------
mplmanager.c      MPMD    A basic "hello world" type program
mplworker.c
```

This directory includes the following files:

- README.mpl (program listing p.73)
- make.mpl (program listing p.76)

- mplmanager.c (program listing p.78)
- mplworker.c (program listing p.81)
- user.cleanup (program listing p.83)
- user.setup (program listing p.84)

To run the sample program in batch under ** IBM MPL **, follow the instructions given in README.mpl (program listing p.73)

5 Compiling message passing programs

The Unix utility "make", or a variant of it, is by far the easiest way to control the compilation and linking of your programs. The user is responsible for describing dependencies between programs in a specially formatted file call a "makefile". In this file, the user can also set which compilers to use, which libraries to include for proper linking, how to "clean-up" directories, and so on. To get started with make, copy a sample makefile from the Examples directory (two are reproduced here) and modify it to suit your needs.

Note that in the two makefiles shown here, different compilers are specified (see the lines beginning with `FC =` and `CC =`). MPL programs require that the mpxlf or mpcc compiler be used, while PVMe programs require only the standard xlf or xlc compiler.

- Sample MPL program makefile (program listing p.48)
- Sample PVM program makefile (program listing p.64)
- Sample PVMe program makefile (program listing p.66)

6 Using the LoadLeveler batch control system

To access nodes of the NIST SP2, outside of the three-node interactive pool, users must submit their jobs to the batch system, IBM LoadLeveler. NIST is currently running Version of 1.3 of LoadLeveler. The commands used to interact with the batch system given below.

6.1 Batch commands

llsubmit *command.file* Submit a job to the batch system. The argument to this command is the name of a file containing special LoadLeveler commands which describe the user's program and machine requirements.

llq [**-l**]

Query a job's status in the batch queue.

llstatus Query the status of the node pools.

llcancel *batchid* Kill a job that appears to be hung (on the queue or while processing) or is otherwise not performing as expected. The argument to this command is the four digit job identifier for the process to be cancelled, and can be extracted from the output of the "llq" command.

llclasses Display a current list of the available batch classes on the SP2.

classes Local version of ¡it¿llclasses¡/it¿ giving a short listing of available batch classes.

For more details on each of these commands, see the local LoadLeveler documentation (http://danube.nist.gov/ll.html)

6.2 Batch job preparation utilities

Since both PVM(e) and MPI typically require the user to play an interactive role in the set-up of the parallel environment, a job submitted through Loadleveler *must mimic the steps a user would take in an interactive environment.* This is accomplished via a shell script, which is invoked when the LoadLeveler job becomes active on the queue. This shell script can be user-defined, but for routine usage, the user may benefit from using prewritten generic scripts. These scripts make use of information provided in the LoadLeveler command file to invoke the appropriate environment control tool (either POE (MPI) or PVM), set the correct number of processors, copy files to/from the local node address space, and start the user's MPMD or SPMD program(s) with the correct arguments and/or input redirection. The scripts do require that certain program information be correctly specified in the LoadLeveler command file, so a user interface has been provided which automates the generation of this command file. This utility is invoked by typing `extoll` at the Unix prompt in the directory where your executables are stored. (Note that this utility superceeds "X-llcreate" and "llcreate".) The program prompts the user for specific information about the parallel program, and then creates an appropriate command file which includes the appropriate generic script file to properly execute the user program on the nodes. For more information, see the Extoll documentation (p.26).

7 Batch job utilities: Extoll

7.1 Introduction to Extoll

Extoll provides a convenient "point-and-click" style interface for generating a LoadLeveler command file matched to a user's specific parallel program. (Note that Extoll superceeds the utilities X-llcreate/llcreate, providing more robust and streamlined services.)

Note: when using `Extoll`, your DISPLAY environment variable must be set to your working console. To accomplish this, type "setenv DISPLAY console.internet.address:0.0" at the unix prompt on danube, and "xhost +danube.nist.gov" at the unix prompt on your console.

When Extoll is invoked, the user is prompted for information about the communication library used (PVMe, MPL, or Oak Ridge PVM), the style of their program (SPMD or MPMD), arguments or input required, batch class and number of processors, and whether a custom setup or cleanup script is required. Finally, when the "Create *.cmd File" button is pressed, a command file is generated based on the input information.

7.2 Use of Extoll

To use `Extoll`, simply invoke it in the directory where your executables are stored (this is often the same as your source directory, or `~/pvm3/bin/RS6K`). check the appropriate boxes and fill in requested information in the entry panel. The script is relatively generous in interpreting your input; for example, the program is insensitive to the case of most responses. In fact, most input can be handled with a click of a button, eliminating the need to worry about typos in the input.

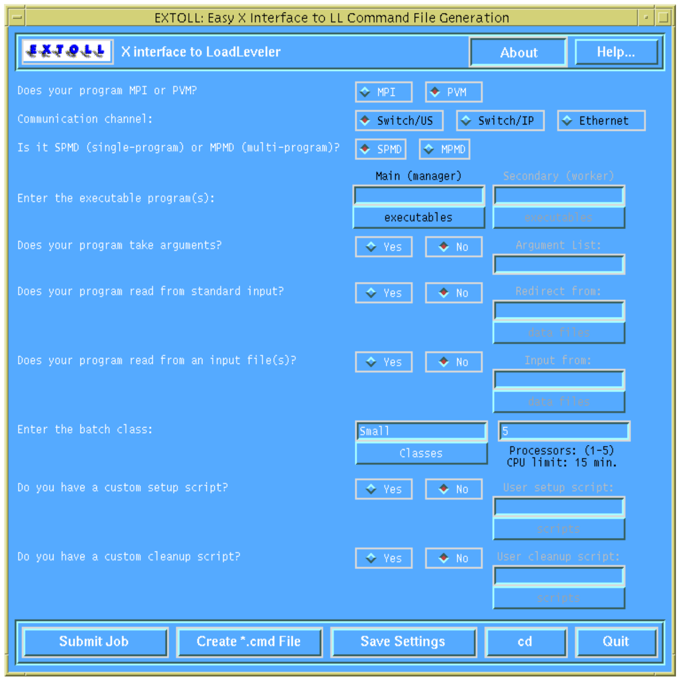

Figure 1: A sample view of the Extoll interface

Of course, with the use of this utility, you maintain the flexibility of editing the generated command file, which could be used, for example, to correct any mistyped program names, etc. You may wish to experiment with various combinations of responses to the queries, studying resulting command file, to help your understanding of how information is relayed to the generic script. If you run into difficulty because your particular program doesn't fit any of the standard scenarios handled by `Extoll` and the appropriate generic script file, please send mail to *kremington@nist.gov* and we will try to update the scripts to accommodate your needs.

A short (8-page) user's guide is available, see ¡A HREF="http://math.nist.gov/acmd/Staff/KRemington/Primer/extoll-doc.ps"¿ Extoll: Easy X-interface to LoadLeveler for Parallel Jobs on the NIST IBM SP2¡/A¿, or the file /usr/local/apps/EXTOLL/doc/extoll-doc.ps on danube.

7.3 Generic Batch Scripts for Message Passing Programs

The Extoll utilitly relies on the use of generic batch scripts, one for each of the available message passing libraries. These scripts are intended to be system supported and maintained, so that all users benefit from locally developed improvements in batch script handling. Actual support of these scripts may vary. Please contact the author at kremington@nist.gov for current information.

7.4 Sample command file

```
#!/bin/ksh
# ----------------------------------------------------------------
# LoadLeveler Command Script:  main.cmd
# Automatically generated at:  Fri Aug 28 14:20:21 EDT 1998
# EXTOLL V. 2.0
# ----------------------------------------------------------------
#
# @ job_name = main
# @ initialdir = /danube/nist/karin/EXTOLL/bin
# @ output = $(initialdir)/$(Cluster).out
# @ error = $(initialdir)/$(Cluster).err
# @ notification = Error
# @ notify_user = karin@danube.nist.gov
# @ class = Small
# @ job_type = parallel
# @ min_processors = 4
# @ max_processors = 16
# @ environment = CLUSTER=$(Cluster); LL_JOB=TRUE
# @ restart = no
# @ checkpoint = no
# @ requirements = (Adapter == "hps_user")
# @ queue
#
# ----------------------------------------------------------------
#    Set up and start an SP2MPI PVM job which uses the SP2
#    switch in User Space mode (high performance).
#
#    NOTES:  > Executables must be compiled according to the
#              PVM documentation in README.mp.
#            > Processes are limited to one per node, and each
#              pvm_spawn generates an "extra" process.
#
```

```
#    (Please see PVM documentation in README.mp for more info.)
# ------------------------------------------------------------------
#    This stub is to set PVM_ROOT and PVM_ARCH in your .cshrc file.
#    See the EXTOLL documentation for proper .cshrc setup.
#
cp $EXTOLL_ROOT/bin/.pvm-switch.stub $HOME/.llpvmstub
# ------------------------------------------------------------------
#
#    Create a PVM hostfile with the Executable path set to /tmp/karin
#             ( using switch addresses hssw*.nist.gov)
#
cat /tmp/$LOADL_STEP_OWNER.$LOADL_STEP_ID.hostfile |
    sed -e 's/grand/hssw/' >> $CLUSTER.hostfile
# ------------------------------------------------------------------
export PVM_ROOT=$HOME/pvm3
export PVM_ARCH=SP2MPI
export PVM_PROCESSOR_LIST=$LOADL_PROCESSOR_LIST
export STARTHOST=`hostname | sed -e 's/grand/hssw/'`
export LOADL_PROCESSOR_LIST=`echo $LOADL_PROCESSOR_LIST |
                             sed -e 's/grand/hssw/g'`
export MP_PROCS=4
export MP_RMPOOL=1
export ARGUMENTS=""
export STANDARD_IN=""
export FILES_IN="my.data"
# ------------------------------------------------------------------
#    Copy executables and 'cd' to working directory:
# ------------------------------------------------------------------
for host in $PVM_PROCESSOR_LIST
do
  rsh $host "ksh -c '[ -d /tmp/karin ] || mkdir /tmp/karin'"
  rsh $host "cd /danube/nist/karin/EXTOLL/bin; cp -f main worker /tmp/karin"
done
cd /tmp/karin
# ------------------------------------------------------------------
#    Now, invoke the daemon and run the [setup] program [cleanup]:
# ------------------------------------------------------------------
$PVM_ROOT/lib/$PVM_ARCH/pvmd3 -n$STARTHOST $CLUSTER.hostfile &
sleep 15
/danube/nist/karin/EXTOLL/bin/user.setup
main $ARGUMENTS < $STANDARD_IN
/danube/nist/karin/EXTOLL/bin/user.cleanup
# ------------------------------------------------------------------
#    Delete executables and other files from working directory:
```

```
# ----------------------------------------------------------------
cd /danube/nist/karin/EXTOLL/bin
for host in $PVM_PROCESSOR_LIST
do
  rsh $host 'rm -r /tmp/karin"'
done
# ----------------------------------------------------------------
#    Finally, halt the daemon and copy back the PVM log file:
# ----------------------------------------------------------------
$PVM_ROOT/lib/$PVM_ARCH/pvm <<EOJ
halt
EOJ
export myuid
myuid=`id -u`
cp /tmp/pvml.$myuid $CLUSTER.pvmlog
echo Batch job completed.
# ----------------------------------------------------------------
```

8 Interactive access to the NIST SP2

8.1 Introduction

The current SP configuration allows users to interactively access the front-end machine (danube) and a small test pool of four nodes (grand17-grand20). This interactive pool should be used to facilitate debugging, or to study performance with interactive analysis tools. To execute large parallel jobs using the SP2 nodes, users must submit their jobs through a batch system, which partitions the machine into node pools and controls access to these pools so that jobs are single-threaded.

Using the interactive nodes is much like using a workstation cluster. You begin by logging onto one of the interactive nodes (grand17, grand18, grand19 or grand20) and then setting up the parallel environment you wish to use. ORNL PVM, IBM's PVMe and IBM's POE(MPL) are all available on the interactive nodes. ORNL PVM may be of particular interest for use with various third-party performance tracing tools, such as XPVM, which are not compatible with PVMe.

8.2 Managing message passing programming sessions

An interesting problem is posed by the existence of two separate and incompatible libaries both sharing the name PVM (ORNL PVM, and IBM PVMe). Without proper care, it becomes quite difficult to manage directories, makefiles, etc. so that PVM and PVMe programs can coexist, and you as a user can switch between one environment and the other without substantial effort. Care is also required to ensure that the interactive environment does not interfere with any batch processing you may have on queue.

These problems can be addressed on the NIST SP2 with two strategies:

- First, the process of setting up your interactive environment can be automated to ensure that all necessary environment variables are set, and that the appropriate PVM daemon and console commands, consistent with the desired environment, are enlisted. This automation is accomplished by appending several lines to your .cshrc file which have been designed to enable switching from one environment to the other during a single interactive session by typing simple commands: "startpvm", "startpvme", or "startmpl".

- Second, batch environments can be designed in a self-contained way, so that they are impervious to anything going on in the user's interactive session (or in their .cshrc file). This is ensured with the use of the generic batch scripts provided to work in conjuction with the batch job preparation utilities X-llcreate and llcreate. If you require the use of a completely customized batch script, and it currently relies on your .cshrc file, you may wish to modify it so that it is self contained, that is, so that all of the variables required by PVM(e) are explicitly set in the batch script, rather than being inherited from the parent environment. The generic scripts can be used as a guide in making any necessary modifications.

8.3 Debugging PVM(e) programs

PVM(e) provides simple facility for spawning processes under a standard debugger. If the pvmfspawn call provides "PVMDEBUG" as the second argument (or "PvmTaskDebug" for spawns in C programs), an xterm window running the spawned process under a debugger will be generated for each spawned process.

This is not by any means a "graceful" debugging environment, but can be used to get a "quick-and-dirty" view of the processes. To invoke the debugger (for PVM or PVMe programs), first compile your programs with the -g compiler option, and set up your PVM(e) environment by issuing "startpvm" or "startpvme" following by starting the daemon. Then, instead of simply executing your main programs, type

debug ⟨main_program⟩

Debug has been aliased to a system supported script which check that your DISPLAY variable is set and that the appropriate debugger scripts are available to PVM(e). The debug script is based on the serial dbx debugger; for help on available dbx commands, type "man dbx" on danube.

8.4 Tutorial examples

The directory /usr/local/doc/interact contains two short tutorials on interactive use of the PVM, PVMe, and MPL environments, and demonstrates how to switch from one environment to another. The tutorial directory includes the following:

- README.FIRST (program listing p.40)
- mpl
    - README.debug (program listing p.43)
    - README.mpl (program listing p.46)
    - make.mpl (program listing p.48)
    - mplmanager.f (program listing p.50)
    - mplworker.f (program listing p.52)
- pvm+pvme
    - README.debug (program listing p.54)
    - README.pvm (program listing p.57)
    - README.pvme (program listing p.60)
    - make.pvm (program listing p.64)
    - make.pvme (program listing p.66)

- manager.f (program listing p.68)
- worker.f (program listing p.71)

9 Some tips on debugging parallel programs

1. Make sure the serial code is bug-free!

   - To this end, it is a good practice to write programs (whenever possible) in a style which can be run on one processor. We include a conversion to SPMD example (p.10) to illustrate how a typical MPMD program pair can be converted into a SPMD program.
   - This step should eliminate any basic indexing errors, and ensure the serial program logic is correct.
   - In Fortran, use `IMPLICIT NONE`, if you don't already. This helps catch mistyped variable names by flagging them as undeclared.

2. Debug your PVM code on your own workstation, using multiple tasks, but a single host.

   - This step will make a coarse pass at eliminating message passing logic errors.
   - Examine code carefully, looking for:
     - Unmatched `send`'s and `recv`'s
     - Message tags that don't match
     - Non-unique message tags; that is, messages which aren't properly distinguished. (This is a problem especially when receiving with wildcard (`-1`) parameters.)
     - Messages that are unpacked upon receipt in a different order than they were packed before sending. The ordering must be identical.
   - Use `pvm_catchout()` to route node stderr/stdout to your console window. (Note: This is not available with the current release of PVMe on the SP2. Node output from PVMe is automaticaly routed to stdout of the main process.)

- Use the `PVMDEBUG` (Fortran) or `PvmTaskDebug` (C) parameter in the `spawn` call. This generates an xterm running "dbx" for each spawned process. (Make sure code is compiled with `-g`.)

  Sample Fortran call:

```
 call pvmfspawn('simplenode', PVMDEBUG, '*',
*                             (nprocs-1), tids, info)
```

  Sample C call:

```
rc = pvm_spawn("simplenode", (char**)0, PvmTaskDebug, "",
                                        (nprocs-1), tid);
```

3. If possible, debug your PVM program on multiple interactive workstations.

   - See the section on interactive access to the SP2 (p.31) for information on how to use debugging on the 4-node interactive pool.
   - If possible, test to see whether the program behaves differently on the interactive nodes of the SP2 than it does on a cluster of non-SP2 workstations.
   - If the program runs on a cluster, but fails regularly or sporadically on the SP2, then the bug is most likely to be highly sensitive to communication timing, and therefore *difficult to track down.* You may wish to seek the assistance of the scientific consulting staff (975-2968).

Finally, *when all else fails*...

4. Try debugging through the batch queue on the SP2 itself. (This feature will be available with an upcoming version of the X-Loadlgen utility.)

   - Make sure that your program(s) are compiled with the `-g` compiler option to `xlf` or `xlc`.
   - See the `llcreate` documentation (p.??) for help in generating a loadleveler command file which will enable debugging. Copy the `debugger` and `debugger2` scripts to your working directory.
   - Submit as usual, *but only when the queues are empty and you are free to wait by your console for you program to run.*

10 Converting an MPMD program to SPMD

There are at least two important advantages to programming in a SPMD style:

- **Debugging** - The program can easily be run sequentially, thus facilitating an important layer of debugging.

- **Efficiency** - Since each node on the SP2 under PVMe can run only one process, (a notable restriction of PVMe which is described in Choosing a Message Passing Library (p.11)), the manager part of a manager-worker program requires its own dedicated node, therefore eliminating one from the pool of nodes available to do the distributed work. In the SPMD form, the manager process spawns worker, and then goes on to do its own share of the distributed workload - thus making more efficient use of the available node pool.

We will assume in this discussion that PVM(e) is being used as the message passing library, however, the same strategy applies equally well to MPL programs.

The key to writing a manager-worker type program in SPMD style is to include a branch in your program just after a `pvm_mytid()` and `pvm_parent()` call, based on whether the particular program instance was spawned (== has a parent). If the program detects that it has no parent, it takes the role of the manager process and does all the start-up functions, including spawning worker tasks and distributing work. It may set aside a portion of the parallel work for itself, so that it does not sit idle while the parallel calculations are carried out. When finished with its share of the calculations, it can then receive messages back from the "worker" processes, perhaps containing the results of the parallel computations.

If the program detects that it was spawned from another process, then it bypasses all the start-up functions, and goes right to work on it's share of the global problem, then possibly sends to the "manager" process results of its calculations.

Following is an example of how a Fortran MPMD style program was transformed into a SPMD program:

The MPMD style program in
manager.f (program listing p.123)

and

worker.f (program listing p.130)

is converted to the SPMD program

spmd.f (program listing p.132)

See the directory /usr/local/doc/Examples/spmd for the complete spmd example, including a makefile and README file with instructions for building and executing the program.

11 Batch debugging

Work in progress...

12 XPVM program monitoring on the SP2

12.1 Program visualization with XPVM

The tool XPVM, developed at Oak Ridge National Laboratory, allows for viewing trace information generated by PVM programs. (Note that PVMe programs do not generate trace information which can be used with XPVM.) XPVM can be used on the interactive nodes, or in the batch environment.

12.2 Requirements

To use XPVM, your program must spawn processes with the "PvmTaskTrace" option. Further, if your program has *any* input requirements, or if you want to monitor a job running in batch under the system pvm.script file, you will need to include a subroutine call in your program to enable it to attach to a pre-established XPVM session at run-time. (This is due to limitations of XPVM, which cannot spawn programs with input requirements and which makes assumptions on the executable directory where your program resides.) Programs with no input requirements may be spawned from within an XPVM session on the interactive nodes, and do not require this extra subroutine call. To cover the most general case, we will assume that the program to be monitored cannot be spawned from with XPVM.

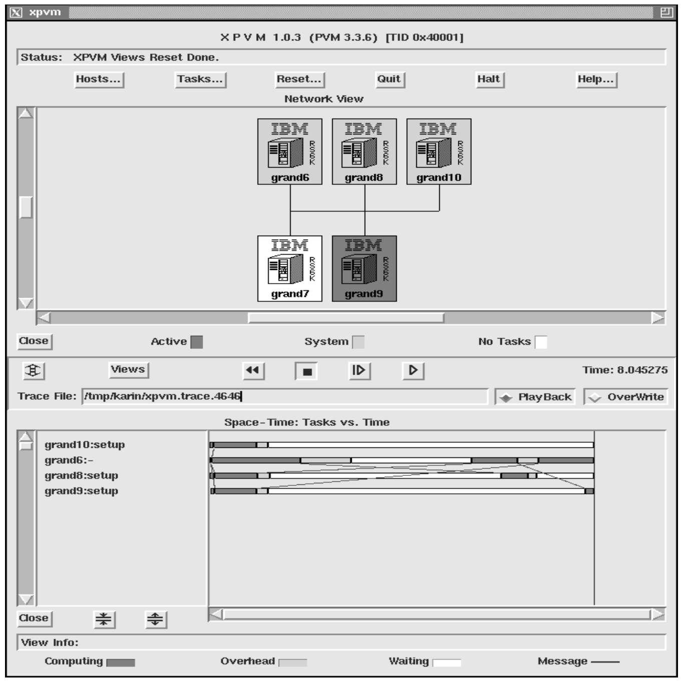

Figure 2: A sample view of the XPVM console window

12.3 Tutorial example: Matrix transposition

The tutorial included in the directory /usr/local/doc/Examples/transp demonstrates the use of XPVM to monitor a SPMD program running in the batch environment. The SPMD program setup.c generates a 3-D matrix distributed across the nodes, and calls a Fortran routine (in transp.f) to perform a matrix transposition. Since both C and Fortran are used, this example also serves to demonstrate how to link multi-language programs. A simple timer is also included, and can be used to get an estimate of elasped wallclock time.

- README.pvm (program listing p.137)
- make.pvm (program listing p.141)
- realtime.c (program listing p.143)
- setup.c (program listing p.145)

- transp.f (program listing p.149)
- user.cleanup (program listing p.154)
- xpvmconnect.c (program listing p.155)

Note the following:

- Your DISPLAY variable must be set on the node running the batch job. The easiest way to do this is to include your DISPLAY setting in your .cshrc file.
- The "grand" node running your batch process must have permission to display on your console. Since it can be difficult to anticipate which specific node will run a batch job (particularly in the case of TEST classes), you may want to add all "grand" nodes to your xhost list on your workstation console.
- Use extreme care when submitting a job to the batch system with the XPVM monitoring option. Interaction is required to properly terminate the XPVM session. If you are not at your console when the job is run, the XPVM will remain open for the maximum time permitted for the batch class which it is running under.

12.4 XPVM on the interactive node pool

To run this tutorial, copy the source and README files listed above to your own area and build the executable program (setup) just as you would for the batch tutorial. Note the arguments required for the exectuable program. (See README.pvm for details.) To use XPVM on the interactive node pool, first follow the intructions for setting up an interactive PVM session, since it is probably easiest to start XPVM *after* you've successfully started PVM. To proceed with XPVM:

- Open *two shell sessions (windows)* running on one of the interactive nodes.
- Check to ensure your DISPLAY is set and that the interactive grand node has permission to open on your console.

- Then, in one of the interactive shell sessions (windows), execute /usr/local/bin/xpvm. You need not specify a hostfile, since the job which is being traced will add hosts to XPVM as processes are spawned. After XPVM successfully starts, and you've selected a tracefile (or accepted the default tracefile), switch to the second window.

  It may seem like a more natural approach to start XPVM as a background process, but pecularities in the XPVM environment (and the fact that it generates its own shell) typically cause this approach to fail.

- In the second interactive shell session (window), execute the program by typing its name and arguments. The included call to xpvm_connect() will cause its trace information to be displayed in the XPVM display area. When the program finishes, you may switch to "playback" mode, and replay the trace file and examine various "views" of the collected information.

- When finished, you may choose to "quit" XPVM, which leaves the PVMvirtual machine intact, or "halt" XPVM, which "halts" the PVM daemon as XPVM is exited. You should choose "quit" if you intend to continue using PVM after leaving XPVM, and "halt" if you are finished with your PVM session.

Index

B
batch command file, 28
batch commands, 25
batch job preparation, 25
batch scripts, 28
batch, 8, 24

C
cmd file, 28
command file, 26, 28
compiling, 24
Cornell Theory Center, 9

D
debugging parallel programs, 33
debugging PVM(e) programs, 31
DISPLAY environment variable, 26
distributed-memory, 10
documentation, 8

E
Extoll, 26

G
generic batch scripts, 28
global reduce operations, 11

I
InfoExplorer, 9
info, 9
interactive access, 30
interactive tutorial, 32
interactive, 8

L
llcreate, 26
LoadLeveler, 24

M
make, 24
man-pages, 8
managing message passing sessions, 31
message passing libraries, 11
mpcc, 24
MPL, 12, 18
MPMD vs. SPMD programming, 35
MPMD, 10
mpxlf, 24
multi-language programs, 37

N
node characteristics, 7

P
performance monitoring, 36
program visualization, 36
programming tools, 36
PVMDEBUG, 33
PVMe, compatibility with PVM, 12, 16
PVMe, , 12
PvmTaskDebug, 33
PVM, 11, 12

S
single threaded, 8
SP node characteristics, 7

SP Switch, 8
SPMD advantages, 35
SPMD example, 36
SPMD, 10

T
timing, 37
tools, 36
tutorial on MPL (C), 23
tutorial on MPL (Fortran), 23
tutorial on PVM(e) (C), 17
tutorial on PVM(e) (Fortran), 16

U
usage policy, 8

W
WWW pointers, 9

X
xlc, 24
xlf, 24
xllcreate, 26
XPVM, 36