```
        /* Receive the edge-weight matrix */
        if (rcv(MATRIX_TYPE) < 0)
        {
            pvm_perror("rcv");
            exit(1);
        }
        if (getndfloat(matrix, SQR(nodes)))
        {
            pvm_perror("getndfloat");
            exit(1);
        }

        /* Compute the appropriate row
           of the new matrix */
        comp_short_paths(matrix, nodes,
                         inum, newrow);

        /* Send the new row back
           to the master */
        initsend();
        if (putndfloat(newrow, nodes))
        {
            pvm_perror("putndfloat");
            exit(1);
        }
        if (snd("loapsp_master", 0,
                ROW_TYPE))
        {
            pvm_perror("snd");
            exit(1);
        }
    }

    /* Leave PVM */
    exit(0);
}
```

```
void comp_short_paths(matrix,
            nodes, row, newrow)
    double *matrix, *newrow;
    int nodes, row;
{
    int i, j;
    double min, tmp;

    for (i = 0; i < nodes; ++i)
    {
        min = matrix[row * nodes + i];
        for (j = 0; j < nodes; ++j)
        {
            if ((matrix[row * nodes + j]
               >= 0.0) &&
               (matrix[j * nodes + i]
               >= 0.0))
            {
                tmp =
                 matrix[row * nodes + j]
                   + matrix[j * nodes + i];
                if ((tmp < min) ||
                      (min < 0.0))
                {
                    min = tmp;
                }
            }
        }
        newrow[i] = min;
    }
}
```

`loapsp_slave.c` contains the source code for the node (or slave) program. **loapsp_slave** receives consecutive stages of the result matrix from the host program, computes new values for one row of the matrix each time, and sends that row back to the host program.

```
/* loapsp_slave.c
 *
 * Brian K. Grant
 * LLNL
 * Numerical Mathematics Group
 * 28 May 1992
 *
 * Purpose:
 *    Compute the lengths of all-pairs
 *    shortest paths for a single row of
 *    an edge-weight matrix each time the
 *    matrix is sent to the process.  The
 *    matrix need be sent to each slave only
 *    log2(n) times, due to the type of
 *    algorithm used (n is the number of
 *    nodes in the graph).
 */

#include <stdio.h>
#include <pvmuser.h>

/* Max component name length */
#define MAXCNLEN 32

/* Find the square */
#define SQR(x) ((x) * (x))

/* Type used when receiving
   the number of nodes */
#define NODE_TYPE 0

/* Type used when
   receiving the matrix */
#define MATRIX_TYPE 1

/* Type used when sending
   rows to the master */
#define ROW_TYPE 2

extern void comp_short_paths();

main(argc, argv)
    int argc;
    char *argv[];
{
    char slave[MAXCNLEN];
    int inum;

    double *matrix;
    int nodes;
    double *newrow;

    int nloops;
    int i;

    /* Perform PVM initialization,
       get instance number */
    if (whoami(slave, &inum) < 0)
    {
        pvm_perror("whoami");
        exit(1);
    }

    /* Receive the number of
        nodes of the graph */
    if (rcv(NODE_TYPE) < 0)
    {
        pvm_perror("rcv");
        exit(1);
    }
    if (getnint(&nodes, 1))
    {
        pvm_perror("getnint");
        exit(1);
    }

    matrix = (double *)
        malloc(sizeof(double) * SQR(nodes));
    newrow = (double *)
        malloc(sizeof(double) * nodes);

    /* Compute the appropriate
       row each time the new matrix
       is received */
    nloops = (int) (log((double) nodes) /
                log(2.0) + 1.0);
    for (i = 0; i < nloops; ++i)
    {
```

```
    if (*file == '\0')
    {
        /* Read from stdin */
        pFin = stdin;
    }
    else
    if ((pFin = fopen(file, "r"))
                == 0)
    {
        return (0);
    }

    if (fscanf(pFin, "%d", &nodes) != 1)
    {
        return (0);
    }
    *pnodes = nodes;
    *pmatrix = (double *)
       malloc(SQR(sizeof(double) * nodes));

    for (nodes = 0;
           nodes < *pnodes; ++nodes)
    {
        for (i = 0; i < *pnodes; ++i)
        {
            if (fscanf(pFin, "%lf",
                &((*pmatrix)[nodes *
                   (*pnodes) + i])) != 1)
            {
                return (0);
            }
        }
    }

    fclose(pFin);
    return (1);  /* success */
}
```

```
/* print_matrix
 *
 * Parameters:
 * The matrix of edge weights:
 *  double *matrix;
 * The number of nodes in the graph:
 *  int nodes;
 *
 * Returns:
 * Nothing.
 *
 * Purpose:
 *      Print the matrix.
 */
void print_matrix(matrix, nodes)
    double *matrix;
    int nodes;
{
/* Index variables: */
    int i, j;

    for (i = 0; i < nodes; ++i)
    {
        for (j = 0; j < nodes; ++j)
        {
            printf("%lf ",
               matrix[i * nodes + j]);
        }
        printf("\n");
    }
}
```

```
    /* Send each process the matrix
       and wait to receive the new
       rows during iteration, until
       the computation is complete. */
    nloops = (int) (log((double) nodes) /
                log(2.0) + 1.0);

    for (i = 0; i < nloops; ++i)
    {
        /* Broadcast the matrix */
        initsend();
        if (putndfloat(matrix, SQR(nodes)))
        {
            pvm_perror("putndfloat");
            leave();
            exit(1);
        }
        if (snd("loapsp_slave",
             PVM_TO_ALL, MATRIX_TYPE))
        {
            pvm_perror("snd");
            leave();
            exit(1);
        }

/* Receive the rows as they are sent
   by the slaves. Each slave processes
   the same row as its instance number: */
        for (j = 0; j < nodes; ++j)
        {
            /* Receive a row */
            if (rcv(ROW_TYPE) < 0)
            {
                pvm_perror("rcv");
                leave();
                exit(1);
            }

            /* Determine which row it is */
            if (rcvinfo((int *) NULL,
               (int *) NULL,
               (char *) NULL, &minum))
            {
                pvm_perror("rcvinfo");
                leave();
                exit(1);
            }

            /* Update the matrix */
            if (getndfloat(
                 &matrix[minum * nodes],
                   nodes))
            {
                pvm_perror("getndfloat");
                leave();
                exit(1);
            }
        }
    }

    /* Output the final results */
    print_matrix(matrix, nodes);

    /* Leave PVM */
    leave();
    exit(0);
}

/* get_matrix()
 *
 * Parameters:
 * Name of the file from which to read:
 *  char file[];
 * Pointer to the edge weight matrix:
 *  double **pmatrix;
 * Pointer to the number of nodes
 * in the graph:
 *  int *pnodes;
 *
 * Returns:
 * 1 if successful, 0 if there was an
 * error in reading the data
 *
 * Purpose:
 *      get_matrix() reads the edge weight
 *      matrix from the file given on the
 *      command line.
 */
int get_matrix(file, pmatrix, pnodes)
    char file[];
    double **pmatrix;
    int *pnodes;
{
/* Input file: */
    FILE *pFin;
/* Number of nodes in the graph: */
    int nodes;
/* Index variable: */
    int i;
```

```
main(argc, argv)
    int argc;
    char *argv[];
{
/* Name of master process: */
    char *master;
/* Instance number of master process: */
    int inum;
/* Number of processors used: */
    int nprocs;

/* Matrix of edge weights: */
    double *matrix;
/* Number of nodes in the graph: */
    int nodes;

/* Number of iterations required: */
    int nloops;
/* Instance number of sending process: */
    int minum;
/* Index variables: */
    int i, j;

    /* Usage: loapsp_main [file] */
    if (argc > 2)
    {
        fprintf(stderr,
           "Usage: %s [filename]\n",
            argv[0]);
        exit(1);
    } else if (argc == 2)
    {
        if  (! get_matrix(argv[1],
               &matrix, &nodes))
        {
            fprintf(stderr,
            "%s: Bad data file \"%s\"\n",
                argv[0], argv[1]);
            exit(1);
        }
    }
    else
    {
        if  (! get_matrix("",
              &matrix, &nodes))
        {
            fprintf(stderr,
                "%s: Bad data from stdin\n",
                    argv[0]);
            exit(1);
        }
    }

/* Enroll with PVM */
master =
   (master = rindex(argv[0], '/'))
         ? master + 1 : argv[0];
if ((inum = enroll(master)) < 0)
{
    pvm_perror("enroll");
    exit(1);
}

/* How large is our
   virtual machine? (just for fun) */
if (pstatus(&nprocs, (int *) NULL))
{
    pvm_perror("pstatus");
    leave();
    exit(1);
}
printf("%s: using %d processors\n",
         master, nprocs);

/* Initiate the subprocesses.  We
    know that the inums will be 0
    to (nodes - 1). */
for (i = 0; i < nodes; ++i)
{
    if (initiate("loapsp_slave",
          (char *) NULL) < 0)
    {
        pvm_perror("initiate");
        leave();
        exit(1);
    }
}

/* Send each process the number
   of nodes in the graph */
initsend();
if (putnint(&nodes, 1))
{
    pvm_perror("putnint");
    leave();
    exit(1);
}
if (snd("loapsp_slave",
    PVM_TO_ALL, NODE_TYPE))
{
    pvm_perror("snd");
    leave();
    exit(1);
}
```

[13] Postel, J. B., "User Datagram Protocol," RFC 768, ISI, August 28, 1980.

[14] Rigsbee, Peter A., *PVM Message-Passing Library Interface,* Draft Version 3.0, Cray Research, Inc., April 17, 1992.

[15] Skjellum, A. and A. P. Leung, *Zipcode: A Portable Multicomputer Communication Library atop the Reactive Kernel,* Proc. of the Fifth Distributed Memory Computing Conf., IEEE Press, pp. 767-776, April 1990.

[16] Sun Microsystems, Inc., *XDR: External Data Representation Standard*, RFC 1014, June 1987.

[17] Sunderam, V., Personal Correspondence (e-mail), July 1992.

The following papers describe software which has collectively been called "PVM 1.0," which is not available or supported and which differs from the currently available software. Some of these reports are available on **netlib**.

[18] Geist, G. A. and V. S. Sunderam, *Network-Based Concurrent Computing on the PVM System*, Summer 1990. Also appeared in *Concurrency: Practice & Experience*, 4(4), pp. 293–311, June 1992.

[19] Sunderam, V. S., *PVM: A Framework for Parallel Distributed Computing*, Summer 1989. Also appeared in *Concurrency: Practice & Experience*, 2(4), pp. 315–339, December 1990.

[20] Sunderam, V. S. and G. A. Geist, *The PVM System: Supercomputer-Level Concurrent Computation on a Network of IBM RISC System/6000 POWERstations*, *Scientific Excellence in Supercomputing: The 1990 IBM Contest Prize Papers*, Volume 2, K. Billingsley, H. Brown, and E. Derohanes, The Baldwin Press, 1992.

A C Program Listing

In this section, we describe the master ("host") and slave ("node") programs for the LOAPSP example, described in section 9.

`loapsp_master.c` contains the source code for the host (or master) program. It initiates the **loapsp_slave** processes, and broadcasts consecutive stages of the result matrix to them.

```
/* loapsp_master.c
 *
 * Brian K. Grant
 * Lawrence Livermore National Laboratory
 * Numerical Mathematics Group
 * 28 May 1992
 *
 * Purpose:
 *   Given a matrix of edge weights,
 *   this program computes the lengths of
 *   the shortest paths between all pairs of
 *   vertices.
 */

#include <stdio.h>
#include <pvmuser.h>

/* Find the square */
#define SQR(x) ((x) * (x))

/* -1 means broadcast to all
   instances of the same name */
#define PVM_TO_ALL (-1)

/* Type used when broadcasting
   the number of nodes */
#define NODE_TYPE 0

/* Type used when broadcasting
   the matrix */
#define MATRIX_TYPE 1

/* Type used when receiving
   rows from the slaves */
#define ROW_TYPE 2

extern int get_matrix();
extern void print_matrix();
```

  - One saves messages and plays them back like ISIS,
  - Another allows checkpoint/restart under PVM 2.4,
- Message receipt selectivity based on message source is promised in 3.0 to supplement type selectivity.

Each of these projects attempts to build on top of PVM 2.4 rather than modifying the basic PVM software. This modularity is meant to extend portability and expandability. Reports are available on request from the PVM developers.

Many alternatives to PVM exist, a very few of which we note here:

- Reactive Kernel / Cosmic Environment (Caltech)
- ISIS (Cornell / Ohio St.)
- Network Express (Parasoft)

Over this past three months, about fifty systems and libraries were identified as supporting workstation-class distributed computing. Several appear to be multi-person-year efforts. We include our list of such products in an appendix to [9].

To get a copy of the full report [9], when available, please send electronic mail to *skjellum@llnl.gov* including your name, electronic mail address, and postal mail address.

11 Acknowledgements

We wish to thank J. Dongarra and G. A. Geist of Oak Ridge National Laboratory (ORNL) for prompt feedback on the first draft version of this report. The first author interacted significantly with V. Sunderam (Emory University), G. A. Geist, and R. Manchek (also ORNL) via e-mail during the course of this study. Their time, advice, and clarifying information are all appreciated. T. Huntsberger (University of South Carolina), H. Simon (NASA Ames), and D. Forslund (Los Alamos), all received the first draft of this document for critical review prior to this final publication; we appreciate their additional feedback. S. G. Smith (LLNL) and A. P. Leung (Syracuse University) provided significant help in proofreading and in clarifying potentially ambiguous statements.

References

[1] Beguelin A., J. Dongarra, G. A. Geist, R. Manchek, and V. Sunderam, *A Users' Guide to PVM: Parallel Virtual Machine*, Oak Ridge National Laboratory Technical Report ORNL/TM-11826, July 1991.

[2] Beguelin, A., J. Dongarra, G. A. Geist, R. Manchek, V. S. Sunderam, *Graphical Development Tools for Network-Based Concurrent Supercomputing*, Proc. of Supercomputing '91, IEEE, pp. 435–445, 1991.

[3] Beguelin, A., J. Dongarra, G. A. Geist, Robert Manchek, K. Moore, R. Wade, J. Plank, V. S. Sunderam, *HeNCE: A Users' Guide: Version 1.2*, Univ. of Tenn. Computer Science Dept, CS-92-157, February 1992.

[4] Crabtree, J. D., and A. Gordon, *Investigation of Parallel Programming Methods,* Dept. of Mathematical and Computer Sciences, Colorado School of Mines, 1992.

[5] Dongarra, J. D. and E. Grosse, *Distribution of Mathematical Software via Electronic Mail,* CACM 3(5), 403–407, July 1987.

[6] Geist, G. A., M. T. Heath, B. W. Peyton, and P. H. Worley, *A Users' Guide to PICL – A Portable Instrumented Communication Library,* Oak Ridge National Laboratory, ORNL/TM-11616, May, 1992.

[7] Geist G. A., Personal Correspondence (e-mail), July 1992.

[8] Geist G. A., Personal Correspondence (e-mail), August 1992.

[9] Grant, B. K. and A. Skjellum, *The PVM Systems: An In-Depth Analysis and Documenting Study*, LLNL, September 1992.

[10] Heath, Michael T., and Jennifer A. Etheridge, *ParaGraph: A Tool for Visualizing Performance of Parallel Programs,* Oak Ridge National Laboratory, May 20, 1992.

[11] Manchek, R., "PVM version 2.4.1 distribution (patch level 1)," July 1992.

[12] Postel, J. B., DoD standard Transmission Control Protocol. RFC 761, ISI, January, 1980.

This is the input file representing the graph to be processed. The first number in the file is expected to be the number of nodes of the graph. Each element of the matrix which follows represents an edge weight. For example, the value in row 1, column 2 represents the weight of the edge connecting node 1 to node 2. Thus, the diagonal is zero and the matrix is symmetric in the case of an undirected graph such as this one. -1 is used to signify the lack of an edge between two nodes.

```
5
 0.0  1.0  5.0 -1.0 -1.0
 1.0  0.0  3.0 -1.0 -1.0
 5.0  3.0  0.0 10.0 -1.0
-1.0 -1.0 10.0  0.0  7.0
-1.0 -1.0 -1.0  7.0  0.0
```

Figure 3: Example LOAPSP Program Input

What follows is the output produced by LOAPSP given the above input data. The matrix is in the same format as the one used for input. Each element represents the length of the shortest path from the node of the graph corresponding to the element's row in the matrix to the node corresponding to its column. If there were no path between two nodes of the graph, a -1 would appear in the appropriate position in the matrix.

```
loapsp_master: using 5 processors
0.00000 1.00000 4.00000 14.00000 21.00000
1.00000 0.00000 3.00000 13.00000 20.00000
4.00000 3.00000 0.00000 10.00000 17.00000
14.00000 13.00000 10.00000 0.00000 7.00000
21.000000 20.00000 17.00000 7.00000 0.00000
```

Figure 4: Example LOAPSP Program Output

  - The PVM system is an evolving research code,
  - No semaphores (could be used with concurrent I/O over NFS),
  - No message-passing contexts (*cf*, [15]),
  - No receipt-selectivity based on source (type only),
  - Single buffer send/receive limitation,
  - Several key issues of distributed computing are not addressed well:
    * Load balancing (none),
    * Fault tolerance (master-slave, redundancy, recovery),
    * Scalability (linear spawn & virtual-circuit broadcast, recursive doubling),
    * Performance (UDP, number of message steps).
  - No ability to trace execution/debugging support,
  - No support for or good hooks for checkpoint-restart.

We conclude that PVM is interesting as a learning tool about distributed computing, and for connecting a few workstations together on simple applications with extremely low communication needs relative to their computational or (local) disk input-output requirements. HeNCE 1.2 may be a better route to painless distributed computing, yet this does not seem to be catching on.

Some limitations noted above should be removed in the 3.0 version; other support software is promised for the 2.4.1 version as well [8]:

- **Xab** should be available shortly for both 2.4.1 and 3.0, It allows realtime as well as ParaGraph views of a PVM run. Debugging features are being added,
- A "high-level library" is being developed to supply global functions to PVM programs,
- A shared-object environment has been built on top of PVM 2.4 that allows C++ programs to share objects transparently,
- A dynamic load balancing package is under development that applications can be set up to utilize,
- Two separate efforts in the area of fault tolerance are underway:

- KSR1, Kendall Square
- I860, Intel RX Hypercube
- IPSC, Intel IPSC/2
- NEXT, NeXT
- PMAX, DEC/MIPS architecture (3100, 5000, *etc.*)
- RIOS, IBM/RS6000
- SGI, Silicon Graphics IRIS
- SUN3, Sun 3
- SUN4, Sun 4, 4c, Sparc, etc.
- SYMM, Sequent Symmetry
- TITN, Stardent Titan
- UVAX, DEC/MicroVax
- VCM2, Thinking Machines CM-2 (Vax front-end)

For parallel machines (such as CM-5), support is to the front-end processors.

9 An Example PVM Program in C

This sample program is dubbed LOAPSP (Lengths Of All Pairs' Shortest Paths). Given a matrix of edge weights of a directed or undirected graph, LOAPSP will produce a matrix specifying the lengths of the shortest paths between all pairs of nodes of the graph. This is not the type of coarse-grain application for which PVM was designed; however, a small example should prove helpful to new PVM users.

The **loapsp_master** initiates one **loapsp_slave** process per node in the given graph. In addition to the source code to LOAPSP, a sample PVM `hostfile`, a `makefile`, an input file, and the corresponding output file are included to complete the example (see figures 2, 1, 3 and appendix A). The sample `hostfile` in figure 1 was used to specify the directory containing **loapsp_master** and **loapsp_slave**. It also dictated which hosts to include as part of the parallel virtual machine. The example graph is described in terms of edge weights to the program. The input file corresponding to this graph is in figure 3. The entire program listing appears in appendix A.

```
# These machines are operating under
# a shared filesystem
* dx=/usr/local/pvm-2.4/etc/pvmd
# ep=report/pvm-demo
athena.llnl.gov
helios.llnl.gov
mercury.llnl.gov
odysseus.llnl.gov
zeus.llnl.gov
```

Figure 1: Example PVM Hostfile

```
all: loapsp_master loapsp_slave

loapsp_master: loapsp_master.c
    cc -L/usr/local/pvm-2.4/lib \
        -I/usr/local/pvm-2.4/include \
        -o loapsp_master loapsp_master.c \
        -lpvm -lm

loapsp_slave: loapsp_slave.c
    cc -L/usr/local/pvm-2.4/lib \
        -I/usr/local/pvm-2.4/include \
        -o loapsp_slave loapsp_slave.c \
        -lpvm -lm
```

This `makefile` was used to compile the example on our cluster. `/usr/local/pvm-2.4/lib` was the location of `libpvm.a`.

Figure 2: Example PVM Host and Node Program Makefile

10 Conclusion

PVM is an evolving system with strong and weak points. Here is our summary.

- Pros of PVM
  - PVM is free and easily obtained via electronic mail,
  - PVM is small, simple, easy to install, run, and understand,
  - PVM is portable across Unix-compatible machines,
  - Heterogeneity is well supported,
  - The large user base likes PVM.
- Cons of PVM

called `enroll()` or `whoami()`, must wait for that process to initialize. The client process would have its `c` and `vw` flags set until the other process completed its initialization.

Finally, a process enrolling itself with the local PVM dæmon or undergoing initialization may temporarily have its `ul` flag set. Also, a process which attempts to enroll with the local PVM dæmon using a different version of PVM than the PVM dæmon uses will have its `ul` flag set. Furthermore, the attempt to enroll will fail and the process will be marked as having exited.

6.5.5 How To Terminate Runaway Processes

The `kill` command can be used to terminate processes. `kill` takes the process group name and instance number as arguments. The selected process is marked as exited and is killed.

6.5.6 Back To The Beginning

The `reset` command kills all running client processes. This begins a new epoch. It takes no arguments.

6.5.7 "Which Processes Are Blocked?"

The `barr` command generates a list of existing barriers and events. The name and quorum of each existing barrier is listed. The number of processes that have attempted to participate in each barrier is given as well. Finally, each event generated by a call to `ready()` is listed.

7 Handling Errors and Tracing Execution

When an error occurs in one of the client library routines, `pvm_errno` is set to reflect the nature of the error. The `pvm_perror()` call will print an appropriate error message to standard error, depending upon the value of `pvm_errno`. The string argument passed to `pvm_perror()` will be incorporated into the message. Usually, it will be the name of the library routine which failed.

If you desire more information about the progress of your PVM programs, you may wish to compile **pvmd** with the `-DDEBUG` compiler flag. This is the extent of PVM's support for tracing program execution. When PVM and HeNCE were designed, it was perhaps decided that the tracing support would be incorporated into HeNCE, rather than into the PVM system. However, it is argued that **Xab** provides the tracing capability needed by PVM, and that no such division of functionality was ever intended. Since we have not been able to use **Xab** yet, we cannot comment further on this program [8].

8 Implementation

Several different protocols are employed on different levels to accomodate the communication needs of the PVM system. Dæmon initialization is performed through a human-readable-character-stream protocol. Most of the **pvmd**-to-**pvmd** communication relies upon a protocol that uses UDP as the underlying transport protocol [13]. Another protocol is used for interaction between client processes and their local PVM dæmon. This protocol uses UNIX-domain datagram sockets to perform interprocess communication, if they are available. Otherwise, it uses UDP over the localhost loopback. Finally, client-to-client messages are sent using both of the above protocols as the transfer mechanism.

Other features of interest are as follows:

- Communication goes from the local (client) process to the local **pvmd**. This data is transferred to the remote **pvmd**, which in turn forwards it to the remote process. Thus, three messages are used to accomplish a non-virtual-circuit transmission.
- The only logarithmic-complexity algorithm employed is a recursive-doubling broadcast,
- Linear instances of **rsh**s are used to spawn **pvmd**s,
- Streams are maintained between the slave **pvmd**s and the master **pvmd** to support parallel printing, and the detection of **pvmd** deaths.

PVM-supported architectures (PVM name, machines) are as follows:

- AFX8, Alliant FX/8
- BFLY, BBN Butterfly TC2000
- C2MP, Convex
- CM2, Thinking Machines CM-2 (Sun front-end)
- CM5, Thinking Machines CM-5
- CRAY, Cray
- HP9K, HP 9000 (Snake)

are separated by colons, and a percent sign ('%') expands into the architecture of the remote host. Once again, relative paths are relative to the user's home directory on that host. The default execution path is `pvm/%`.

Note that if the password must be entered for any of the hosts in the virtual machine, then **pvmd** must be executed in the foreground. An example `hostfile` is given with the sample program in section 9.

6.4 Epochs

The PVM dæmons keep information about exited processes, and it is still possible to request information on such processes. When all processes have exited, all of the dæmons are notified of the beginning of a new epoch. The process information of the previous epoch is kept by the master dæmon, and is available on request (see section 6.5.4). Otherwise, the new epoch provides an entirely clean slate (group names and instance numbers may be reassigned). The duration of the execution of a PVM program is usually one epoch.

6.5 Interactive Commands

When **pvmd** is executed with the `-i` command-line option, the master PVM dæmon runs in interactive mode. It prints the prompt "`pvmd,` " and accepts commands from standard input. The currently available commands are detailed below.

6.5.1 "I Quit!"

The `quit` command is used to terminate the program. Under normal circumstances, **pvmd** exits quietly. Typing the end-of-file character will also cause **pvmd** to exit. If one of the PVM dæmons is interrupted, it outputs the virtual machine configuration, its local process list, its external process list, its event list, and its barrier list and then exits. When any of the PVM dæmons die, all of the PVM dæmons kill their local client processes and terminate.

6.5.2 "Help!"

The `help` command, `h`, and `?` all refer to the help command. `help` can be used to receive information about any of the **pvmd** commands. `help` may be followed by the command about which information is desired. `help` is the same as `help help`.

6.5.3 "What Is The Virtual Machine Configuration?"

The `conf` command is used to list the configuration of the virtual machine. The host number, host name, architecture type, IP address, **pvmd** UDP port number, and maximum UDP message length are given for each host of the virtual machine (*cf*, section 8). Also, the **pvmd** version number, `libpvm.a` version number, the maximum message-fragment length, and whether or not client messages are XDR-encoded is outputted.

6.5.4 "Which Processes Are Still Running?"

The `ps` command is used to list the processes of the current or previous epoch. By default, `ps` lists the processes of the current epoch. The `-o` option is used to specify that the processes of the previous epoch should be listed instead. The component name, instance number, host, and flags of each process are printed. The flags are explained in the help message for `ps`. The flags indicate whether a process is running, has enrolled, has exited, has died, has just died, is calling the local **pvmd**, is waiting for a client message, is waiting on an event, is waiting at a barrier, is waiting for a virtual circuit port number, or is using a PVM library of a different version than the version of the PVM dæmons.

Some of these flags have esoteric meanings, which are covered in more detail in section 8. One example is the `jd`, or just died, flag. The `jd` flag indicates that a process is *freshly dead*. As soon as an initiated PVM client process terminates, the parent **pvmd** is notified and the process becomes freshly dead until the parent PVM dæmon can deal with the death of its child. At that time, the process is marked as having exited. The `d` flag is *never* used.

The `r` flag is set for processes which the PVM dæmons believe are running. This may not be the case for processes which are not spawned by the PVM dæmons, since the PVM dæmons are not notified when such processes die. The programmer is responsible for ensuring that `leave()` is called before these processes exit.

A process waiting for a message from another client process via either the `rcv()` or `rcvmulti()` routine would have its `c` and `mw` flags set. The client sends a request for a message to the local **pvmd** and blocks while waiting for a response. Similarly, a process having called `waituntil()` or `barrier()` would have its `c` flag and `ww` or `bw` flag set while it waited, respectively.

A client attempting to send a message over a virtual circuit to an initiated client, which has not yet

To broadcast a message to all processes:

```
double data[100];
int toinum = -1;
int msgtype = 1;
initsend();
putndfloat(data, 100);
snd("nodeprogram", toinum, msgtype);
```

Do not attempt broadcast on virtual circuits; it's not efficient.

To get information about the most recently received message:

```
int len;
int msgtype;
char fromname[32];
int frominum;
rcvinfo(&len, &msgtype, fromname,
          &frominum);
```

Barriers/Events To implement a barrier wait with `n` processes:

```
int n = 10;
barrier("catchup", n);
```

To wait on an event:

```
waituntil("eventname");
```

To generate an event:

```
ready("eventname");
```

Analogous Fortran calls are supported for the entire PVM library.

6.2 The PVM Dæmons

The PVM dæmons are responsible for a majority of the inter-host communciation occuring in the system. They also store most of the state needed by the system, such as the virtual-machine configuration and client-process information. Furthermore, the master PVM dæmon provides the user with a mechanism by which information regarding the running system can be obtained.

The command **pvmd** is used to start the master PVM dæmon. The master dæmon consults the provided `hostfile` to determine how to configure the virtual machine. It subsequently spawns the slave dæmons on all of the hosts listed in the `hostfile`. The master PVM dæmon prints a message once all of the PVM slave dæmons have been initialized. If an error occurs while the master **pvmd** attempts to spawn a slave **pvmd**, a message describing the problem will be printed. If a message such as `expected ``pvmd'' got ``...''` is printed, then one of your login scripts (*e.g.*, `.cshrc`) probably produced some output when the slave **pvmd** was executed. Such output interferes with the communication between the master and slave PVM dæmons.

The `hostfile` must be specified on the command line. The user may also use the `-i` option, which indicates that **pvmd** should operate in interactive mode. If no arguments are specified, then `pvmd` assumes that it is a slave and begins the slave initialization protocol described in [9]. The `hostfile` format is discussed below.

6.3 Hostfile Format

Each line of the `hostfile` is either blank (containing whitespace – spaces or tabs – only), a comment, sets default options, or contains a hostname followed by a list of options. Comments begin with a `#`, and may be preceded by whitespace. Blank lines and comments are ignored. To set the default options, one begins a line with $*$, possibly preceded by whitespace, and lists the options after it whose default values are to be changed. This changes the default options for the hosts which follow that line in the `hostfile`. The default options may be changed more than once in the `hostfile`. All other lines must begin with a host name, optionally preceded by whitespace, and are followed by a list of options for that host. The options, covered below, are separated by whitespace.

- `pw` indicates that `rexec()` should be used when spawning a slave **pvmd** on that host so that a password may be entered by the user. The default action is to use **rsh**, which must be located in `/usr/ucb`.
- `lo=` may be used to specify an alternate login name other than the login name of the user which owns the master **pvmd** process. An example would be `lo=grant`.
- `dx=` followed by the pathname of the **pvmd** command redefines the location of the **pvmd** to execute on a host. Relative paths are relative to the user's home directory on that host. The default path of **pvmd** is `/tmp/pvm/pvmd`.
- `ep=` is used to define a new execution path. When a client program calls `initiate()` or `initiateM()`, the specified path is searched for the program to execute. Directories in the list

To receive a message:

```
double data[100];
int msgtype = 1;
rcv(msgtype);
getndfloat(data, 100);
```

or

```
int ntypes = 3;
int types[] = { 1, 5, 4 };
int msgtype;
msgtype = rcvmulti(ntypes, types);
switch (msgtype)
{
        case 1:
                ...
        case 4:
                ...
        case 5:
                ...
}
```

Note, there is no `initrecv()` analogous to the `initsend()` documented below.
To receive a message over a virtual circuit:

```
int idata[10000];
int msgtype = 4;
vrcv(msgtype);
getnint(idata, 10000);
```

or

```
int ntypes = 2;
int types[] = { 1, 2 };
int msgtype;
msgtype = vrcvmulti(ntypes, types);
```

To check for a message:

```
int msgtype = -1;
if ((msgtype = probe(msgtype)) >= 0)
{
        /* There is a message */
        ...
}
```

or

```
int ntypes = 4;
int types[] = { 7, 2, 9, 5 };
int msgtype;
msgtype = probemulti(ntypes, types);
```

Sending Data To prepare to send data:

```
/* default conversion on send: */
    initsend();
/* specify conversion: */
    int conv_option;
    pvm_inithow(conv_option);
```

Use of `initsend()` implies conversion of specified data using XDR format [16]: if *all* the hosts use the same internal data format, no conversion takes place. For `pvm_inithow()`, `conv_option` can be either `PvmDataRaw` or `PvmDataDefault`. In the former case, data transferred by these routines is not converted. In the latter case, `pvm_inithow()` acts exactly as `initsend()`.
To add data items to a message (with automatic conversions for heterogeneity, in the usual case):

```
/* short integer: */
    putnshort(short_data, n_shorts);
/* integer: */
    putnint(int_data, n_ints);
/* long integer: */
    putnlong(long_data, n_longs);
/* floating point: */
    putnfloat(float_data, n_floats);
/* double precision: */
    putndfloat(double_data, n_doubles);
/* complex float: */
    putncplx(complex_data, n_complexs);
/* complex double: */
    putndcplx(dcomplex_data, n_dcomplexs);
/* null-terminated string: */
    putstring(str);
/* byte-stream data: */
    putbytes(byte_data, n_bytes);
```

To send a message:

```
int idata[10000];
int toinum = 11;
int msgtype = 4;
initsend();
putnint(idata, 10000);
snd("nodeprogram", toinum, msgtype);
```

To send a message over a virtual circuit:

```
int idata[10000];
int toinum = 11;
int msgtype = 4;
initsend();
putnint(idata, 10000);
vsnd("nodeprogram", toinum, msgtype);
```

6.1 C Library Calls

PVM 2.4.1 provides primitives for process-creation, process-termination, message-passing, data-marshaling, system-inquiry, barrier-synchronization, and event-handling.

Enrolling/Unenrolling To enroll a program with PVM:

```
int inum;
if ((inum = enroll("program")) < 0)
        pvm_perror("enroll");
```

To notify PVM that a program is exiting:

```
leave();
```

`Enroll` need not be called in a PVM-initiated program if `whoami()` is the first PVM call in such a program (see below). The call to `leave()` may be omitted as well. This is due to the fact that PVM-initiated processes are children of the PVM dæmons, which are notified when such processes die. See also section 6.4.

Starting/Killing Node Programs To start a program:

```
nodeinum = initiate("nodeprogram",
                            "RIOS");
```

or

```
nodeinum = initiateM("nodeprogram",
                    "mercury.llnl.gov");
```

To kill a program:

```
int nodeinum;
terminate("nodeprogram", nodeinum);
```

Information Accessors To get the component name and instance number of a PVM process (and to initialize PVM for an initiated process):

```
char name[32];
int inum;
whoami(name, &inum);
```

To obtain information about the virtual machine:

```
int nhosts;
int nidfs;
pstatus(&nhosts, &nidfs);
```

To check the status of a PVM process:

```
int inum = 3;
if (status("component", inum))
{
        /* It is running */
        ...
}
```

The `pvm_mstat()` routine can be used to obtain more detailed information about the virtual machine. The arguments to `pvm_mstat()` are most easily described by presenting an example.

```
int i;
int nhosts;
int *ifprocs;
char **hosts;
char **arches;
    .
    .
pvm_mstat(&nhosts,&ifprocs,
            &hosts,&arches);
for (i = 0; i < nhosts; ++i)
{
    printf("Host %d is ", i);
    printf("%s of type %s\n",
        hosts[i], arches[i]);
    printf("It %s have PVM ",
       ifprocs[i] ? "does" : "doesn't");
    printf("processes running.\n");
}
```

Receiving Data To extract data items from a received message (with automatic conversions for heterogeneity):

```
/* short integer: */
    getnshort(short_data, n_shorts);
/* integer: */
    getnint(int_data, n_ints);
/* long integer: */
    getnlong(long_data, n_longs);
/* floating point: */
    getnfloat(float_data, n_floats);
/* double precision: */
    getndfloat(double_data, n_doubles);
/* complex float: */
    getncplx(complex_data, n_complexs);
/* complex double: */
    getndcplx(dcomplex_data, n_dcomplexs);
/* null-terminated string: */
    getstring(str);
/* byte-stream data: */
    getbytes(byte_data, n_bytes):
```

- PVM 2.4.1 treats parallel computers as "atomic entities,"
- PVM-initiated programs may be executed on parallel front-ends (hosts),
- PVM calls may not meet the needs of tightly coupled applications.

While PVM 1.0 documents indicate a PICL emulator, syntactic support for PICL was not provided in version 2.4.1. The PVM 1.0 strategy included a more obvious portability path to and from multicomputer codes.

5 PVM 1.0 Summary

In the foregoing, we have in several circumstances noted the differences between the currently supported PVM, version 2.4.1, and its ancestor, version 1.0. To summarize:

- Dæmons were completely symmetric and somewhat fault tolerant,
- Only initial communications between two clients went through **pvmd**s, subsequent communications were direct,
- No support for virtual circuits was provided,
- Shared memory and mutual exclusion were implemented,
- **xpvm** provided a graphical user interface (GUI) for debugging and managing execution of PVM programs,
- PICL interface provided the ability to monitor performance and trace execution,
- Efficient architecture- and network-specific implementations existed.

We note that in the significant user feedback collected about PVM, users repeatedly are asking for version 1.0 features to be reincorporated into future system releases.

PVM 3.0 Preview As part of the evolution of PVM, Cray Research has become involved in its future specification, and implementation on the Cray MPPs and vector machines. Other vendors are also supplying input for PVM 3.0. As of now, the following qualitative differences are proposed (subject to change; *cf*, [14]):

- Different names for all and different parameters for most library routines,
- Processes identified by a unique number,
- Dynamic process groups (replaces fixed components),
- Separate broadcast routine (to process groups),
- Barriers instantiated on a per-group basis,
- Non-blocking receive to replace `probe()`,
- Dynamic host configuration again,
- Capability to send UNIX signals to PVM processes,
- Multiple send and receive buffers.

The latter change (multiple buffers) removes a significant restriction of the existing system. The other features may have less dramatic effect on users. A backward-compatibility feature for 2.4.1-compatible codes is promised, to avoid having users rewrite their codes.

6 Functionality

PVM version 2.4.1 should be portable to any machine running a version of UNIX that supports Berkeley sockets. PVM provides basic message-passing facilities, barrier synchronization, a form of user-defined event handling, and the ability to initiate new processes on demand. PVM is intended to be used for applications which require medium- to coarse-grain parallelism. In fact, applications with very large-grain parallelism (*i.e.*, independent components which occasionally interact) will probably perform very well under PVM. PVM is intended to allow programmers to take advantage of the architectural advantages of different machines, so that each independent component of a program may be concurrently executed on the computer that is best suited to perform that phase of the computation. For example, the user interface to a mathematical tool could run on a graphics workstation while the computation is done by a large single-program, multiple-data (SIMD) system. More generally, the programming model used can either be SPMD (a simple, general-purpose client can be used to spawn functionally identical programs) or full MIMD/MPMD (*e.g.*, a master/slave paradigm)[1].

[1] 'SPMD' stands for single program, multiple data; 'MPMD' stands for multiple program, multiple data, 'MIMD' indicates multiple program, multiple data.

- Distributed locks.

PVM 2.4.1 does not support shared memory; neither system has support for data mapping, data distributions or data replication. The developers thought that shared memory was an interesting research idea, to be tested out in the experimental code, but which was not suitable for the public-release 2.x version.

Distributed I/O under PVM PVM does not really target the issue of distributed input-output very much. Here are key comments:

- Output from slave **pvmd**s and remote client processes is channeled through the master **pvmd**,
- Each line of output is prefixed with the host name,
- PVM 2.4.1 does not support distributed locks for concurrent file access,
- PVM 2.4.1 does not include any scalable I/O concepts.

The serialization of `printf`-like output through the master dæmon is a visible bottleneck when many processes are actively outputting data. Serious applications will clearly need to read and write substantial amounts of data, and with sufficiently high bandwidths that they spend a non-trivial fraction of their time actually computing. No systems of which we are aware really address distributed I/O adequately at this time. PVM does not inhibit the use of any other, existing distributed I/O mechanisms (such as NFS or AFS).

PVM Programming Environment The following are the key aspects of the PVM environment:

- Portability (see section 6),
- Simple, easy-to-learn user interface,
- C and Fortran compatibility libraries,
- UNIX command-line interface to the master **pvmd**.

There are some notable omissions:

- No application or numerical libraries,
- No support for parallel debugging,
- No way to trace execution other than debugging messages,
- No facility for monitoring performance.

In addition, the PVM 1.0 documents describe the following features, not in PVM 2.4.1:

- **xpvm**, support for distributed program management,
- **xdbx**, support for distributed debugging,
- PICL, widely used message-passing interface [6],
- ParaGraph, post-mortem performance analysis [10].

The **xpvm** system provided the following capabilities with PVM 1.0:

- PVM process management interface,
- PVM initiated-program executable file management interface,
- PVM host management interface,
- Parallel debugging interface (using **xdbx**),
- Interface to the ParaGraph execution-tracing tool.

The Paragraph tool provides these additional features:

- Accepts post-mortem trace files generated by PICL library routines,
- Constructs graphical view of execution from trace data,
- Provides many different portraits of execution.

A replacement for **xpvm**, known as **Xab**, is currently in development. No firm public release date was available at the writing of this report. There are indications that an internal version has been made available to a limited audience, but we have not been able to get a copy as yet.

Relation to Multicomputer Message Passing Systems PVM does not deal with the potentially thousands of processors in a multicomputer or multiprocessor. If such a machine is used, a PVM-initiated program resides on the host (or front-end) of such a computer. All information computed in parallel within the parallel machine must be serialized through the front-end program in order to be shared with other PVM programs elsewhere in the network.

- Thus far, no relation to multicomputer internal message passing,

- A basis for network-SCHEDULE (later HeNCE),
- A stable and robust system,
- Enough functionality to support HeNCE,
- Hooks for future enhancements.

An augmented design team worked on PVM 2.0. The systems have somewhat incompatible syntax; many 1.0 features are absent in 2.0. According to our feedback, every 1.0 feature that could be robustly supported in a heterogeneous environment was included in 2.0. Quoting from [8], "Features that would barely work in the lab were left as on-going research issues." We were told by Sunderam [17], prior to [8], that he and Geist had little input into the implementation of the new system; they only required that the interface remained the same or similar.

Above, we mentioned issues that arise in distributed computing. We summarize how PVM 2.4.1 addresses each of these issues.

How PVM Addresses Load Balancing Whereas PVM 1.0 documents describe a form of dynamic load balancing, PVM 2.4.1 has no capabilities in this area. Others have noted this as well; [4] indicates that lack of load balancing is PVM 2.4.1's most serious limitation. Version 2.4.1 has the following features that impact load balancing:

- Hosts are assigned in a round-robin manner,
- Specific hosts may be requested,
- Specific architectures may be requested,
- PVM 2.4.1 knows nothing of the relative speeds or loads of the hosts.

Scalability Issues PVM 2.4.1 is not really a scalable distributed computing environment. In many instances, communication is serialized through dæmons (**pvmd**s), while linear spawning and authentications are used. Consequently, while the system is workable for a few or tens of distributed computers, it will bog down eventually. In summary,

- Slave **pvmd**s are spawned consecutively,
- Recursive doubling is used to broadcast to all PVM dæmons,
- Local communication is linearized through the **pvmd** dæmon process,
- Virtual-circuit broadcasts are performed in a linear fashion,
- Client processes (PVM-initiated programs) can maintain only a small number of virtual circuits.

File descriptor limitations on UNIX workstations further impair scalability because one TCP stream is maintained between the master dæmon and each slave dæmon [12]; for instance, this limits the number of hosts to about 60 on a Sun workstation cluster. Virtual circuits, which circumvent dæmon interactions, have their performance impaired by this restriction as well.

Security Issues Security is not a major concern in the PVM 2.4.1 system. PVM relies entirely on UNIX-based security techniques, and encourages the widespread use of `.rhosts` files to avoid explicit password authentications. The key security properties are as follows:

- A virtual machine is set up on a per-user basis,
- All processes run as user processes,
- Slave **pvmd**s are spawned with **rsh** or `rexec()`,
- PVM provides no mechanism for data encryption.

Fault-Tolerant Behavior While PVM 1.0 documents describe a completely distributed system that had some fault tolerant capabilities, PVM 2.4.1 does not emphasize this at all. It can be argued that addressing fault tolerance well is extremely expensive, but this remains an open question for future systems to address better. In summary,

- The master **pvmd** is crucial,
- If any **pvmd** dies then the system quits,
- No form of checkpoint-restart is supported,
- Processes can check the status of other processes,
- Fault tolerance must be implemented by the user.

Data Distribution/Locality Issues PVM 1.0 documents describe a distributed shared memory, with the following features:

- Typed and untyped shared blocks,
- Automatic data-format conversion applied to typed blocks,

not production quality; they were pure research codes held together with "bubble gum and string."

Since PVM 1.0 was never made available it is not clear why it is compared to PVM 2.4 in this paper. Does the reader gain any useful knowledge by saying that something he has never seen and couldn't use is different from something that is available? Many interesting features were tried in the internal versions of PVM but they were not robust enough to be used by anyone except the developers. Our research is continuing to try to develop such robust features in a heterogeneous environment ...

We chose and retain the format and comparisons alluded to above for several reasons. First, the substantial fraction of literature available refers to the pre-release PVM (so-called PVM 1.0 version). Several users with whom we have interacted claim to have used and/or tried PVM 1.0, so it is difficult for us to eliminate the possiblity that users did write codes on the system. More importantly, however, is that the documents referring to PVM 1.0 remain conspicuously on **netlib**, and most of them have no dates, or distinguishing marks to indicate that they are outdated, or do not apply to PVM 2.4.1. Furthermore, by showing the progression from pre-release PVM to PVM 2.4.1 to PVM 3, we establish a history in the full report. This type of developmental history is to help our local users, and other readers see how the system is developing, and what they should expect in terms of features.

3 Distributed Computing Overview

Distributed computing is coercing autonomous, networked computers to work simultaneously on a set of problems in a cooperative manner. Within this loose definition, many software packages exist to help support distributed computing. We have found that, in addition to many publicly discussed systems, there are myriad ad hoc packages. In general, such packages seek to help users achieve the following goals:

- Better computer performance at a lower cost,
- Utilize *idle* cycles and resources,
- Match hardware to applications.

Within the framework of distributed computing, many issues arise that are not as difficult to address in single-workstation computing. They include:

- Load Balancing,
- Scalability,
- Security,
- Fault Tolerance,
- Data Distribution/Locality,
- Distributed I/O,
- Programming Environment,
- Tracing Execution.

In rating any distributed-computing system, it is interesting to discuss how well, if at all, the system is able to cope with the above issues. Sometimes the local workstation operating systems make addressing these issues arbitrarily difficult, suggesting the need to make workstation operating systems conscious of their distributed computing environment.

In what follows, we specialize our discussion to the PVM system, particularly version 2.4.1.

4 PVM Overview

Given that a large number of ad hoc systems have been in use for some time, the evolution of a public system like PVM was inevitable. The original system's (version 1.0) goals were as follows:

- Execute large parallel applications consisting of relatively independent components with *good* performance,
- Enable separate program components to execute on machines best suited for them,
- Provide a general, flexible, portable, and inexpensive concurrent programming environment,
- Provide a *unified framework* for developing large parallel systems efficiently.

PVM 1.0 was successful inasmuch as its authors were able to address real applications on networks of workstations in a useful way [20].

PVM 2.0 constituted a complete rewrite and change of emphasis compared to the original code. Its main goals were to provide:

to see the limits of performance and capabilities of the system. Furthermore, this information forms the basis for future objective comparisons with other heterogeneous messaging systems and operating environments, that are beyond the scope of this article and the full report.

In [9], we strive to establish PVM's target audience, and then see how well PVM serves the needs of that audience. In this way, we hope to offer useful guidance to potential users of the system. Particularly, we distinguish between the ideas of High-Performance Distributed Computing (HPDC) and generic Distributed Computing (DC). In the former area of research and development, gigabit testbeds will connect large-scale parallel, vector and SIMD (single-instruction, multiple-data) supercomputers at high network speeds, with notions of scalability of data transfer (I/O), memory bandwidth, and performance. In the latter, workstations, and, possibly, front-ends of supercomputers will be connected via ethernet. Issues of heterogeneity of performance and data format are prevalent to both areas of discourse. For now, we can summarize by saying that PVM addresses issues in the DC world, rather than the HPDC world.

2 About this Study

This study was undertaken to provide the Open Computing Facility (OCF) at LLNL with significant, in-depth information about the PVM system as currently available from **netlib** and as documented in various papers on **netlib**, in recent conference proceedings, and in journals. Since the typical user of PVM downloads documents and code from **netlib**, and perhaps has access to the journal articles, we felt that this procedure represented a logical way to begin learning of the system. The first author was assigned this task, under the guidance of the second author. Starting with knowledge of workstations, networks, and protocols, but no previous bias for any type of distributed computing environment or approach, the first author downloaded articles and code from **netlib**, interacted significantly with members of the PVM design team, and discussed PVM with other knowledgeable users throughout the U.S., mainly by electronic mail. Because many millions of dollars will be committed by LLNL alone to codes based on a chosen distributed computing platform, LLNL's OCF and other code developers needed a clear, deep, and critical appraisal of the current and future potential of the PVM system to provide for their needs. It was further decided to add criteria by which further systems could be compared objectively (these are our distributed computing issues, see below). Such critiques remain to be done.

Because we wished to share the products of this report as quickly as possible, a summary version of the full report was prepared first (the first draft of this document). Its title is based on the deliverables of the full report: an in-depth analysis and study of versions 1, 2, and 3 of PVM. The full report, alluded to throughout this document, provides much of the detailed information gleaned from our study; it is still in preparation at this time. The draft of this document was given to selected users for their comments, and to people knowledgeable about PVM for their reactions and criticisms. Most especially, it was given to the designers of PVM at Oak Ridge National Laboratory. The feedback to the draft came from G. A. Geist, and we have made modifications where we agree with his comments and suggestions.

However, to represent the developers' viewpoint on this system properly, which still differs in some respects from ours, we quote at length from their comments. Geist writes:

> PVM is an ongoing research project at Emory University, Oak Ridge National Laboratory, and the University of Tennessee. It is a basic research effort aimed at advancing science, and is wholly funded by research appropriations from the U.S. Department of Energy, the National Science Foundation, and the State of Tennessee. Owing to its experimental nature, the PVM project produces, as incidental deliverables, software that is of utility to researchers in the scientific community and to others. This software is, and has been distributed freely in the interest of advancement of science. While we have and will continue to offer some software support, it is in the spirit that user feedback helps direct future research areas. Our purpose is research not [to be] software suppliers.
>
> The paper spends a lot of time comparing what is termed PVM 1.0 and PVM 2.4.1. What is not known and hence is misrepresented in the paper is that there never was a PVM 1.0. PVM 1.0 (as cited in [18]) is the term the project team coined to represent all the original PVM versions and varieties that were developed internally to experiment with different distributing computing concepts. These versions were never made available since they were developed to empirically validate research ideas and therefore

The PVM Systems: An In-Depth Analysis and Documenting Study – Concise Edition*†

Brian K. Grant,‡ Anthony Skjellum§
Numerical Mathematics Group
Lawrence Livermore National Laboratory
Livermore, CA 94550

Abstract

PVM (Parallel Virtual Machine) is a basic message-passing system for heterogeneous collections of networked computers, developed at Oak Ridge National Laboratory, University of Tennessee and Emory University. This report is our "third-party" effort to answer most common questions about PVM, and to dispel many misconceptions regarding its capabilities. We strive to present an accurate assessment of the system, while adding our own opinions regarding the system, and offering a few suggestions for the future of the system. PVM version 2.4.1, the latest version at the time of the writing of this report, is summarized within this document; our forthcoming full report includes significantly more detailed information. Information on how to get the full report is included.

1 Introduction

PVM (Parallel Virtual Machine) is a basic message-passing system for heterogeneous collections of networked computers, developed jointly at Emory University, University of Tennessee, and Oak Ridge National Laboratory. This report is our "third-party" effort to answer most common questions about PVM, and to dispel many misconceptions regarding its capabilities. We strive to present an accurate assessment of the system, while adding our our own opinions regarding the system, and offering a few suggestions for the future of the system. PVM version 2.4.1, the latest version at the time of the writing of this report, is summarized within this document; the forthcoming full report [9] includes significantly more detailed information.

Be forewarned that other versions of PVM differ significantly from version 2.4.1, both syntactically, and in terms of features supported. Part of the motivation for this report is the lack of sufficient current documentation on PVM from its developers. The confusion that the differing versions in the field add to discussions about PVM as a product, its use, and range of applicability, together with outdated documentation, make it difficult to know exactly what features one should expect or use. We found that much of the available information is obsolete and sometimes conflicting, and therefore subject to misinterpretation. The source of the confusion is that the obsolete documents refer to some older version of PVM, which actually possessed *more* features and support software than the 2.4.1 version does. We have put notes in the bibliography to indicate which documents refer to the currently available software, and those which refer to pre-release software (which is generically referred to as PVM 1.0). Though this is not covered in this short-form report, PVM 3 will also differ syntactically and in some features from PVM 2.4.1. Hence, if you read this document and find you are using PVM 3, this document has by now been rendered obsolete.

The basic syntax and features of PVM 2.4.1 are covered here, while the protocols used by PVM as well as important implementation details will be covered later in [9]. Protocol information will be mainly of interest to application designers who are making long-term decisions about choices of portability platforms. A thorough knowledge of the interactions within the system is valuable when evaluating PVM as a whole. This will help application designers and typical users

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy. Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

†Revision 1.02-16sep92.

‡Supported in part by the Open Computing Facility at Lawrence Livermore National Laboratory.

§Address correspondence to: LLNL, Numerical Mathematics Group, L-316, 7000 East Ave., Livermore, CA 94551, USA. (510) 422-1161. FAX: (510)423-2993; *skjellum@llnl.gov.*