A Qualitative Comparison of Network Linda and PVM

Ann Petrie
and
Ron Kerr
Computing Service
University of Newcastle upon Tyne

30th September 1993
(final revision, July 1994)

Foreword

The bulk of this report was written by Ann Petrie. Her first informal draft was subsequently revised and extended by Ron Kerr for publication in the Parallel Processing Memoranda series published by the Computing Science Department of the University of Newcastle upon Tyne.

The first draft of this report was written in September 1993 and, although revised since then, the technical content is still based on the information gathered at that time. Since PVM and Network Linda are both subject to continuous development, this report may not precisely reflect the current state of either. Indeed, new releases of PVM have appeared but it is not believed that these have significant bearing on the opinions expressed in this report.

Linda is a co–ordination language formed by extending a standard programming language such as FORTRAN or C by a small number of operations designed to enable the description of parallel computations. Linda has been implemented for a variety of parallel systems ranging from integrated multiprocessors to clusters of networked workstations. "Network Linda" is the name given to Scientific Computing Associates' Linda implementation for networked workstations and it is with respect to this product that this comparison has been written. In the interests of brevity and fluency, in some places the term "Network Linda" has been shortened to "Linda".

Contents

01 Introduction

A number of systems now exist which support the parallelisation of programs using a loosely coupled network of workstations.

Two such systems are Network Linda, a commercial product produced by Scientific Computing Associates (SCA), and PVM (Parallel Virtual Machine) which is part of a research project started at Oak Ridge National Laboratory (ORNL) and which is available by anonymous ftp.

A good survey of such environments is provided by Turcotte [T1]. This paper also surveys network systems for batch job distribution. In February 1993 Newcastle University Computing Service installed a 10 node version of SCA's Network Linda on 10 Hewlett Packard 710s and this has been used to run parallel programs written in FORTRAN, C and C++ (and, indirectly, Pascal via automated translation to C).

PVM is one of many alternatives to Network Linda and is already popular because it is public domain software and therefore free of charge. Each of these systems has its advantages and disadvantages and the purpose of this document is to offer guidance to users wishing to select the one more appropriate for their applications.

Because of limited time available to perform this comparison, this report is probably incomplete but it is hoped that it addresses the most important points. The report is based on the principal author's experience in using Network Linda supplemented by:

Study of the PVM User Guide and Reference Manual [G1], P[1].

Study of some of the literature on PVM [D1], [G3], [S6].

Examination and conversion to Linda of a "Super Lattice" PVM program written by Mark Cusack, Department of Physics.

Construction and execution of a Linda job farming harness for neural network experiments.

Examination of a naval architecture PVM program.

Study of the literature on HeNCE (Heterogeneous Network Computing Environment), a graphical parallel computing environment built on top of PVM.

02 Approaches to Parallel Programming

When a problem is solved using a parallel programming system, various decisions have to be made.

How to divide the computation into suitable tasks to be assigned to the participating processors.

How to communicate between the processes that carry out these tasks. (This includes the scheduling and synchronization of processes and the transmission of data between them.)

Both Network Linda and PVM are coarse–grained programming systems in that programs are divided into a number of co–operating components whose computations are substantial relative to the work involved in transferring data between them. However, these environments address the process synchronisation and information exchange in different ways.

Two ways which have been traditionally used for interprocess communication (IPC) in parallel programming systems are message passing and the use of shared memory. In Network Linda, IPC is achieved through the use of a distributed shared global data space called "tuple space". It is distributed in that different parts can reside on different processors but it looks like one global memory space to the component processes. In this sense it can be described as virtual shared memory. In PVM, IPC is achieved through message passing.

03 The Linda Model

The Linda parallel programming model [C1], [C2] was developed as part of a research project at Yale and is one in which processes communicate and synchronize by means of a globally shared associative memory which is referred to as "tuple space". Tuples are collections of fields logically "welded" to form persistent storage items. To some extent, they resemble records in conventional programming terminology.

A process which wishes to communicate information generates a new tuple and adds it to tuple space. A process which needs information looks for it in tuple space, performing an associative search in which an appropriate tuple is located by its field structure and by some of the field values. To update a tuple, the tuple is removed, modified and returned to tuple space. Restricting tuple space modification in this manner creates an implicit locking mechanism which ensures proper synchronization of multiple accesses. In appropriate situations, the inter–process data transmission implied by tuple space operations can be optimised by a broadcast feature.

An active process may delegate a subcomputation by generating a "live tuple". This results in a new process capable of executing concurrently with other processes, including its creator. On completion, the live tuple is transformed into an ordinary tuple, characterised by the results of its computation, and is placed in tuple space, where it can be accessed by other processes.

The Linda tuple space operations are the the following:

out *place a tuple in tuple space.*
in *remove a tuple from tuple space.*
rd *read the values of a tuple in tuple space but leave the tuple there.*
eval *create a live tuple. This starts new process(es) which eventually cause a tuple to be placed in tuple space.*

Tuples are selected by the *in* and *rd* primitives on the basis of their field values. *in* and *rd* are blocking operations in the sense that if no matching tuple exists the calling process is suspended until one arrives. *in* and *rdp* (see below) are non–blocking variants of *in* and *rd*.

Linda is a model for parallel programming rather than a programming language, but the addition of the Linda operations to an existing language such as C or FORTRAN turns the language into a Linda variant that can be used for parallel programming. The language which is produced functions as a co–ordination language and provides the means of combining distinct processes into a complete parallel program. The Linda operations are orthogonal to those in the base language and provide complementary capabilities which enable parallel programming to be carried out.

The programming semantics of Linda are simple and the programmer is spared the chore of explicit message passing.

04 Overview of Network Linda

SCA's (Scientific Computing Associates) Network Linda is an implementation of the Linda parallel programming model which runs on clusters of workstations, not necessarily of the same architecture. Both C–Linda and FORTRAN–Linda compilers are provided. In addition to the four basic Linda operations mentioned earlier, these programming languages have two further operations *inp* and *rdp* which are predicate forms of *in* and *rd*:

inp *non blocking form of in*
rdp *non blocking form of rd*

If a matching tuple can be found in tuple space, these operations behave exactly the same as *in* and *rd* and return a value of 1 (success). However, if no matching tuple can be found, they do not block but return a value of 0 (failure).

Network Linda is capable of supporting heterogeneous configurations. It is available on Sparcs (both SunOS 4.1 and SunOS 5.2), Dec Alphas (OSF 1.3), DECstations, RS/6000s, SGIs, and HP 9000/700 series. Preliminary ports have been done to some other workstations, and other ports have lapsed (Apollo, for example), but the above are the primary platforms. Incompatible data formats are handled by invoking the data conversion option which uses the Sun XDR representation. Simple variables and arrays can be exchanged in this manner but structures require closer attention. Currently, all ports of Network Linda use UDP and therefore need TCP/IP. The latter is provided by the UNIX operating system on which all the current ports run. At Newcastle, Network Linda is licensed for a homogeneous configuration of ten HP 9000/710 workstations.

SCA also produce a version of Linda which runs on distributed memory machines. Currently Network Linda and distributed memory Linda are not able to communicate with each other, but this will change when SCA release something called “Paradise”. This will be a more flexible Linda which will allow tuple spaces to be persistent, and to be accessed from many different programs.

Both C–Linda and FORTRAN–Linda compilers are supplied, the FORTRAN compiler being built on top of the C compiler. There is a Code Development System (CDS) which allows code to be developed and executed in simulated parallelism on a single workstation and there are some very useful debugging tools. Whether a program is compiled to be used with the CDS or to be run over the network depends on the setting of environment variables.

Programs compiled for running over the network are run using a utility *ntsnet* which has responsibilities which include starting and shutting down the program, waiting for normal or abnormal termination conditions during program execution, querying remote systems for load averages and initiating remote processes. The utility *ntsnet* has a large number of options and also uses a number of (optional) configuration files in order to make running a Linda program as flexible as possible. Using the options and configuration files the user can specify (among other things):

the number of machines (nodes) on which a program will run
the relative speeds of machines
the minimum and maximum number of processes per node
the location of remote executables (copying them if necessary)
the location of remote working directories
various parameters relating to scheduling

When a network version of a program runs, the Linda system determines on which machines various processes will execute. If the job requires fewer processors than the total licensed, *ntsnet* selects those which are least heavily loaded at that moment. Normally the user does not control which processor will host a particular process. However, including a processor’s identity among tuples’ fields allows that machine to locate tasks assigned exclusively to itself.

More information about Network Linda can be found in the Reference Manuals and Release Notes [S1], [S2], [S3], [S4], [S5].

05 Overview of PVM

PVM (Parallel Virtual Machine) is an on–going research project started in 1989 at Oak Ridge National Laboratory (ORNL) and now involving people from ORNL, the University of Tennessee and Carnegie Mellon University. The project produces software that is useful to other members of the scientific community and, in the interests of advancing science, this software is available freely over the Internet. Appendix A describes how to acquire the PVM software and documentation.

PVM is a software package that allows a (possibly heterogeneous) group of machines across a (possibly heterogeneous) set of networks to act as a resource for parallel computing.

Under PVM a user–defined grouping of serial, parallel and vector computers appears as one large distributed memory machine. Configuring a personal parallel virtual machine involves simply listing the names in a file when PVM is started. (Additional parameters in this file allow for different login names on different machines and for different locations of various files required by PVM.) This configuration can be changed later if desired either by the use of a special PVM console program *pvm* or by a running program. The *pvm* program also allows users to find out about the status of their parallel virtual machines.

A user's virtual machine can overlap with that of another user and several applications can be run on the same virtual machine. However, a user can have only one virtual machine active at any one time. Applications can be written in C or in FORTRAN and are parallelised by using message passing constructs common to most distributed memory computers. As with Network Linda, the FORTRAN system is built on top of the C system.

PVM supplies the functions to initiate and terminate tasks on a user's virtual machine and allows these tasks to communicate and synchronize with each other. Communication constructs include those for the exchange of data structures as well as high level primitives such as broadcast, barrier synchronization and event synchronization. In particular, PVM handles all message conversion if participating computers use different data representations, for example if they have different byte ordering or different floating point representation. PVM also includes control, tracing and debugging features. A summary of PVM operations is given in Appendix B. More detail can be found in the User's Guide and Reference Manual [G1], [G2].

The PVM system consists of two parts:

A daemon, *pvmd3*.

This resides on the machines making up the virtual parallel computer. It provides, among other things, an inter–host point of contact.

A library of PVM interface routines.

The library contains routines for passing messages, spawning processes, co–ordinating processes and modifying the virtual machine. Applications must be linked with this library to use PVM.

Application programs using PVM may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, the processes in existence may have arbitrary relationships with each other. Furthermore, any process may communicate and/or synchronize with any other.

A PVM application consists of one or more "instances" of one or more "components". A component is a conventional serial program but it will normally have PVM calls in it. Each instance of each component is a process which can be regarded as a subtask of the application.

PVM is available for nearly thirty machine architectures [G1], [G2], [M1]. Usually these are being used with a version of the UNIX operating system, but recent releases of PVM have included versions for non–UNIX systems. However, the instructions for building the recent software releases are written for a UNIX system.

06 Versions of and Documentation for Network Linda and PVM

Both Network Linda and PVM are evolving items of software. New features are continually being added to both systems and additional machines are being supported.

For example, since the installation of Network Linda Version 2.5 by the Computing Service in February 1993 a new release, Version 2.5.2, was received in May 1993.

Version 3.0 of PVM was a major update and enhancement of Version 2.4 in which the names of the PVM library calls were changed to names starting with "pvm" to avoid clashes with names in other libraries. This was also released in February 1993. Since then several further versions have been released, PVM 3.1 (May, 1993), PVM 3.2 (August, 1993), PVM 3.2.2 (September, 1993) and PVM 3.3 (June, 1994).

The documentation for Network Linda is provided by the C–Linda and FORTRAN–Linda guides and the appropriate Release Notes [S1], [S2], [S3], [S4], [S5]. This documentation corresponds to the software that is purchased and therefore tends to be in step with it.

The documentation for PVM can be obtained by anonymous ftp from *netlib2.cs.utk.edu* in directory *pvm3* [G2]. A number of published papers are also to be found there as well as slides for overhead projectors and a useful PVM Quick Reference Card [D1], [G3], [M1], [P2], [S6]. In the recent past, the User Guide and Reference Manual have tended to lag behind the software. However, the current version of the User Guide (*ug.ps*) matches the latest software release 3.3.

A useful recent addition to the PVM documentation at the beginning of September, 1993, was a set of UNIX–style manual pages (see Appendix A). Previously all documentation was in the form of PostScript files which are not easy to consult on–line.

07 Development and Debugging Support

The usual host system profilers and debuggers can be used with both Network Linda and PVM.

In addition, the Linda system provides various tools to assist with development and debugging in a number of ways. These are listed below and described in a little more detail in what follows.

- a Linda Code Development System
- Tuplescope – an X–based visualization and debugging tool
- Post Mortem Analyser
- a utility *pgtrace* for use with ParaGraph

The Linda Code Development System simulates parallel program execution in a uniprocessor environment. During the initial stages of development, this enables problems arising from faulty software and synchronisation operations to be confined to a single machine. To use the Code Development System (CDS) the environment variables LINDA_CLC and LINDA_FLC for C and FORTRAN programs respectively must be set to "cds" rather than to "network" before compilation. If the environment variable is unset, it defaults to "network". The CDS is not currently subject to licensing restrictions and can be run on any machine. CDS executables may initiate the Tuplescope through which process synchronisation and tuple manipulation can be observed and debugging activities performed.

Tuplescope is an X–based visualization and debugging tool. As well as the usual debugger features, such as single step mode and breakpoints, Tuplescope can display tuple classes, data in specific classes and visual indications of process interactions throughout the course of a program run. The native debugger for the machine can also be used with Tuplescope. This means that programs can be observed and debugged at both the Linda process synchronisation level and at the individual FORTRAN/C statement level. The native debugger is initiated simply by clicking on a process icon displayed by the Tuplescope. The Tuplescope debugger is extremely useful and we are not aware of any parallel debugging tool to equal it.

The run time Tuplescope debugger for programs whose parallel execution is simulated on a single machine by the CDS is complemented by a Post Mortem Analyser which simulates tuple space operations based on trace information collected during genuinely parallel runs. The interface to the Post Mortem Analyser is very similar to that of the Tuplescope. However, there is an additional debugging mode, Reverse Execution, which makes it possible to reverse the sequence of tuple space events at any point in the simulated execution. The direction of execution can be changed any number of times and, by scrolling execution backwards and forwards, it is often possible to locate an error by homing in on it without restarting the program from the beginning.

The utility *pgtrace* is used to convert a set of Linda trace files into a single ParaGraph trace file. Linda trace files are produced by C–Linda programs that have been compiled with the –linda profile option. These trace files contain the times at which Linda operations were executed, and at which they completed. ParaGraph is an X11–based visualization tool available from Oak Ridge National Laboratory. Its displays are designed for use with distributed memory, message passing architectures. The *pgtrace* utility performs the necessary post–processing needed to turn Linda operations (such as *out* and *in*), into message passing operations appropriate for ParaGraph.

PVM version 3.0 does not have any additional development and debugging tools other than those supplied with the C and FORTRAN compilers and the operating system. However, all PVM routines return an error condition and two routines (*pvm_perror* and *pvm_serror* for C programs) allow the automatic detection and reporting of errors. In addition, PVM tasks can be started manually under any standard debugger such as *dbx*. PVM tasks that are spawned can also be started under a debugger by setting an appropriate flag in the *pvm_spawn* call. Further details are given in the User Guide [G1], [G2] which recommends the following three steps in trying to debug PVM programs. (These are useful recommendations for debugging any parallel program.)

1 Run the program as a single process and debug as any other serial process. (The purpose of this step is to catch indexing and logic errors unrelated to parallelism.)
2 Run the program using 2 to 4 processes on a single machine. (The purpose of this step is to check the communication syntax and logic.)
3 Run the same 2 to 4 processes across several machines. (The purpose of this step is to check for synchronization errors that are produced by network delays.)

For version 2.4 of PVM, there is a beta test version of a set of tools Xab (X–Window Analysis and Debugging) which provides for the visual analysis and debugging of PVM programs [B2]. The basic idea behind Xab is to instrument every PVM call so that it sends a message to a monitoring process which can either display the event or save it in a file for later analysis.

During the final editing of this comparison document, PVM release 3.3 was announced and, in conjunction with it, version 1.0 of XPVM. This is presumably the successor to Xab but we have not yet studied its features.

Related to PVM is an environment called HeNCE (Heterogeneous Network Computing Environment) [B1], [B3]. This is a parallel programming environment built on top of PVM. Using HeNCE one can describe the parallelism in a program graphically and HeNCE automatically adds the PVM code for parallelisation. HeNCE produces a trace file which can be used to perform a post mortem examination of the execution. More is said about HeNCE later.

08 Ease of programming and suitability for different problems

Although a few papers have been written addressing the performance of PVM and Linda, apart from [C3], there seems to be little presenting a qualitative assessment of these environments.

Apart from the inherent logical difficulty of writing parallel programs, programming in Linda has been found to be easy, especially when the program is written from scratch rather than derived from an existing serial program. Even the parallelisation of serial programs in Linda can be relatively simple if they were well–structured at the outset.

At present we have less experience in using PVM but there are many more things that need to be taken care of than when using the simpler Linda model. The user of PVM has to undertake explicitly many of the, often tiresome, tasks that are performed automatically by the Linda system.

In PVM, the components of messages transmitted between processes must be carefully packed and unpacked. This requires the explicit description of the data representations of the data items involved. In FORTRAN/PVM, the types of data being transmitted must be expressed as explicit parameters to the packing subroutines. In C/PVM, the types of data determine the names of the packing subroutines. Experienced PVM programmers find that this buffer packing is a frequent source of errors. Moreover, subsequent alteration of a data type, e.g. extending precision, entails locating and altering all pack/unpack operations involving that item.

In Linda, inter–process communication is accomplished via higher–level tuples using *out*, *rd* or *in* operations specifying data structures whose representations are deduced automatically by the system.

To take a specific example (from the "Super–Lattice" program) the following set of FORTRAN/PVM operations which creates a message buffer, fills it with data needed by nodes and then broadcasts it:

```
call   pvmfinitsend(0, Info)
call   pvmfpack(integer4, NodeNum, 1, 1, Info)
call   pvmfpack(integer4, Tids, NodeNum, 1, Info)
call   pvmfpack(integer4, XRlvs, NRlvs, 1, Info)
call   pvmfpack(integer4, YRlvs, NRlvs, 1, Info)
call   pvmfpack(integer4, ZRlvs, NRlvs, 1, Info)
```

call pvmfpack(real4, ArrXBulkPt, NBulkPts, 1, Info)
call pvmfpack(real4, ArrYBulkPt, NBulkPts, 1,Info)
call pvmfpack(real4, ArrZBulkPt, NBulkPts, 1, Info)
call pvmfpack(real4, BulkFormFactors, 8, 1, Info)
call pvmfmcast(NodeNum, Tids, 1, Info)

is coded in Linda by the single operation:

out('bulkbase', NodeNum, Tids, Xrlvs, Yrlvs, Zrlvs, ArrXBulkPt,
ArrYBulkPt, ArrZBulkPt, BulkFormFactors)

and the complementary set of operations in PVM to receive and unpack the data:

call pvmfrecv(MTid, 1, Info)
call pvmfunpack(integer4, NodeNum, 1, 1, Info)
call pvmfunpack(integer4, Tids, NodeNum, 1, Info)
call pvmfunpack(integer4, XRlvs, NRlvs, 1, Info)
call pvmfunpack(integer4, YRlvs, NRlvs, 1, Info)
call pvmfunpack(integer4, ZRlvs, NRlvs, 1, Info)
call pvmfunpack(real4, ArrXBulkPt, NBulkPts, 1, Info)
call pvmfunpack(real4, ArrYBulkPt, NBulkPts, 1, Info)
call pvmfunpack(real4, ArrZBulkPt, NBulkPts, 1, Info)
call pvmfunpack(real4, BulkFormFactors, 8, 1, Info)

is coded in Linda by the single operation:

rd('bulkbase', ?NodeNum, ?Tids, ?Xrlvs, ?Yrlvs, ?Zrlvs, ?ArrXBulkPt,
?ArrYBulkPt, ?ArrZBulkPt, ?BulkFormFactors)

It is clear from the above examples that data communication in PVM can be much more messy than in Linda. However, there are situations in which Linda data communication can be more untidy than one would wish, and in which PVM communication, while still messy, enjoys certain advantages. These situations seem to be associated with the communication of arrays involving dynamic memory allocation or of sequences of data items when the sequence length is determined at run time. PVM's superiority in this context stems from the fact that data items are packed individually into the message buffer, a fact which in other contexts can be judged a nuisance.

An example of the dynamic memory allocation situation was revealed to us during the conversion of a PVM C program to Linda. The PVM master process built arrays in dynamically allocated virtual memory and transmitted them to slave processes by packing into the message buffer first the array size and then the data itself. The buffer contained many of these dynamic arrays interspersed with many simple items. Although these simple and composite items had to be packed individually, eventually the buffer represented a single transmission unit, sent and received with single send/receive operations. The equivalent of the send buffer could have been accomplished far more elegantly with a single Linda tuple but emulating the receive and unpack operations posed problems in Linda. The otherwise tedious sequential unpacking of the PVM receive buffer meant that the length of an array was known in time to allocate memory for the data which was unpacked separately next. Since the fields of Linda tuples are atomically and automatically transferred to their destinations, this meant that some ingenuity had to be exercised in order to recover the length of an array before transferring its data. Several possibilities exist but these at best compromise Linda's elegance and expressiveness, and may involve fragmentation accompanied by awkward devices to preserve the logical association between related fragments.

The following C/PVM fragments illustrate the situation. Here, the message buffer contains an item specifying the number of zones in a ship structure, this being used to allocate memory to accommodate the zone characteristics which follow in the same message buffer:

Master process send:

```
bufid = pvm_initsend(PvmDataDefault);
....................................
pvm_pkint(&tot_hull_zones, 1, 1);
for (i = 1; i  <=tot_hull_zones; i++) {
        pvm_pkint(&hull_zone[i].nos_comp, 1, 1);
        pvm_pkint(hull_zone[i].comp_no, hull_zone[i].nos_comp+1, 1);
        pvm_pkdouble(&hull_zone[i].min, 1, 1);
        pvm_pkdouble(&hull_zone[i].max, 1, 1);
}
....................................
pvm_send(tid, ......);
```

Slave process receive:

```
pvm_nrecv(parent_tid, SLAVE_RECEIVES);
....................................
pvm_upkint(tot_hull_zones, 1, 1);
*zne = MYALLOC(ZONE, (*tot_hull_zones)+1);
zone = *zne ;
for (i = 1; i <= *tot_hull_zones; i++) {
        pvm_upkint(&zone[i].nos_comp, 1, 1);
        zone[i].comp_no = MYALLOC(int, zone[i].nos_comp+1);
        pvm_upkint(zone[i].comp_no, zone[i].nos_comp+1, 1);
        pvm_upkdouble(&zone[i].min, 1, 1);
        pvm_pkdouble(&zone[i].max, 1, 1);
}
```

The other situation we discovered can be described quite simply by considering the transmission of a column of an array but much more complicated scenarios are possible. In PVM, the column would be packed into the buffer either using a loop to pack the elements individually or by using an element count and a stride increment. Again, there would ultimately be one message buffer to transmit. Since Linda tuples are created atomically and there is no equivalent of the stride factor, a column of an array would need to be transmitted by constructing a separate tuple for each element. Again, the association and ordering of all the elements of a particular column needs some contrivance. Alternatively, the elements would first have to be transferred to an intermediate aggregate structure. Actually, this specific example is only a problem for C–Linda since FORTRAN–Linda can cope with array slices. However, as said above, the inability to represent a data sequence of dynamic length can give rise to much more complicated problems in Linda and, indeed, the above code fragments also illustrate one of the simpler instances of these in which fields from dynamic arrays of records are transmitted.

It is believed to be the case that anything that can be programmed using Linda can be programmed using PVM and vice versa. However, some things are clearly more easily accomplished using one system than the other. The consequences of the different styles of inter–process communication are illustrated above but, apart from these, there are other ways in which the different approaches can affect things such as the deployment of processors and the style of the parallel logic.

For example, there is no way within a Network Linda *eval* operation itself to force the resultant process to execute on any particular node or system. Adding a functional layer in front of the target routine can accomplish this in a rather contrived manner, as illustrated in the C–Linda User's Guide and Reference Manual [S1], [S2].

```
master()
for (i=0; i < NWORKERS; i++) eval("worker", do_worker());

do_worker()
{
<get the hostname or architecture type via a standard system call>
if (!strcmp(host,"moliere")) worker_1;
elseif (!strcmp(host,"goethe")) worker_2;
elseif (!strcmp(host,"sparc")) sparc_worker;
<and so on>
}
```

In PVM, however, the *pvm_spawn* operation, which starts a new process, has a flag, the value of which determines whether the process starts:

wherever PVM chooses
on a particular host
on a particular architecture

(This same flag also determines whether the process will start up under the debugger and whether it will generate PVM trace data.)

Consequently, forcing a task to be performed on a particular node is much more easily done in PVM than in Network Linda.

Tuples in Linda are anonymous and do not belong to any particular process. They can be removed from tuple space by any process. In PVM, however, a message sent by *pvm_send* is sent to a particular process (it also has a message tag) and the receiving operation *pvm_recv* blocks until a message (with a specified message tag) has arrived from a specified process (PVM also has multicast and broadcast operations *pvm_mcast* and *pvm_bcast* which allow messages to be sent to several processes or to all).

Consequently, writing programs involving multiple server processes which are required to co–operate in an inherently load–balanced manner is much simpler in Linda since any process which is ready at some moment may start working on any outstanding task which it is capable of undertaking. PVM may be more flexible than Linda in the sense that the programmer has more direct mechanical control of the execution. On the other hand, Linda offers a higher level programming model which can simplify the construction of programs with complicated logical interactions. Linda certainly relieves the programmer of distracting details which have no logical bearing on the problem being addressed.

09 Programming a particular problem (Superlattice program)

In order to get the flavour of PVM, a Super Lattice program coded in PVM by one of the physicists, Mark Cusack, was studied. This showed what was involved in writing a complete program and executing it. The program was run on Suns, HPs and a mixture of both.

This program was converted to Network Linda and executed on the HP cluster for which the Computing Service is licensed. The performance of the two programs was very similar with PVM being, on average, slightly faster. The PVM times were also more consistent, i.e. less variable. Every so often a run of the Linda program would give the message:

> Linda: Warning — More evals than processors. Deadlock may result.

On no occasion did it actually deadlock, but, when this message appeared, the elapsed time was considerably longer being of the order of 90 seconds rather than the usual value of between 20 and 27 seconds. (Presumably this was because one or more processor was hosting more subcomputations than the others in a non–load–balanced situation and so their enlarged workloads determined the duration of the computation.)

Due to the shortage of time available to perform this comparative assessment, for ease of conversion the Linda program was structured in the same manner as the PVM one. This also had the advantage that the Linda Tuplescope display (debugging system) mirrored the PVM program. Had there been more time, the Linda program would have been structured differently and this might have speeded it up somewhat.

10 Performance of Network Linda and PVM

Several papers compare the performance of Network Linda and PVM sometimes in combination with other parallel programming systems for workstation clusters [D2], [M2], [M3]. Douglas et al. [D2] which is an update of an earlier paper by Mattson et al. [M3] performed experiments in three communication scenarios.

a 2–node test (a simple ping–pong program measuring conflict–free communication between the nodes)

a 4–node test (a more complicated communication pattern looking at simultaneous shifts of data around a ring of 4 nodes) a commercial molecular dynamics application. (this was only done for the most efficient memory passing paradigm on the first two tests and the virtual shared memory C–Linda

In the first two specialized tests, PVM performed slightly better than Network Linda but the authors' overall conclusion from all the tests was that, as communication patterns become more complex, performance differences between systems decrease substantially and, for an actual application, the differences actually disappear.

Matrone et al. [M2] use three tests which they describe as "the most used algorithms in parallel computing"

broadcasting	data transmission from one node to all others
swapping	data transmission from one node to the neighbours
collecting	data transmission from all nodes but one to the latter

The code for the tests in the Douglas et al. paper can be obtained by anonymous ftp from pub/tr975.tar.Z on casper.cs.yale.edu)

Unlike Douglas et al. [D2] who only measure the time for the actual communication pattern they are studying, Matrone et al. [M2] measure the elapsed time. They note the start–up time overhead for Network Linda of around 25 seconds for all of their tests. (One assumes that the corresponding start–up overhead for PVM takes place when the virtual machine is set up and does not have to be repeated every time a program is run.) Otherwise, they report that PVM and Network Linda appear to have the same performance as regards the net communication time until the message size becomes very large when the performance of PVM degrades. Although they were using much smaller message sizes (800 to 800,000 bytes as opposed to 1 to 6 or 8 Mbytes), Douglas et al. [D2] noted a similar drop in PVM's performance in the 4 node test when the message sizes became large, which they could not understand.

As mentioned earlier, both Network Linda and PVM are continually evolving. For example the papers referred to above all used a version of PVM earlier than 3.0. Consequently any repeat of the tests would be expected to be be slightly different using the current versions.

11 HeNCE (Heterogeneous Network Computing Environment)

HeNCE [B1], [B3] is an X–Windows based software environment, built on top of PVM, which enables users to to develop programs to run on a network of computers. In this environment, users describe their programs graphically in terms of subroutines (nodes) and data dependencies (arcs) and provide (non–PVM) code for the subroutine nodes and information about the data. From this the HeNCE system will create, compile and execute a PVM program. There is no need for users to do any explicit parallel programming.

The HeNCE philosophy of parallel programming is to have the programmer specify the parallelism of a computation and to aid the programmer as much as possible in the tasks of designing, compiling, executing, debugging and analyzing the parallel computation. Central to HeNCE is a graphical interface which is used to perform these tasks.

In HeNCE the programmer specifies graphs, a variant of directed acyclic graphs (DAGs) where the nodes in the graph represent procedures and the edges denote dependencies. These graphs are input by the user using the graphical interface. (There is also a textual interface by which a programmer may create a program graph using a conventional editor). The procedures represented by the nodes can be written in either C or FORTRAN and in many cases can be taken from existing (sequential) code.

An essential feature of HeNCE is the possibility for the user to specify multiple implementations of HeNCE nodes based on the target architecture. At runtime HeNCE dynamically chooses which machine will execute a node and the correct implementation of the procedure is chosen.

Once a HeNCE graph has been written and compiled, it can then be executed on a user–defined collection of machines. The programmer specifies a cost matrix showing the relative costs of running procedures on particular architectures and HeNCE will schedule the program procedures on particular machines taking this cost matrix into account. As it executes HeNCE can display graphically an animated view of its execution.

We have not had time to study HeNCE in sufficient detail but it appears to be an interesting interface/environment. It seems to reinforce our intuitive view that it ought to be possible to describe synchronisation within parallel programs in a manner which is independent of the detail of the computation and, moreover, in a form from which, in theory, parallel code for various languages/environments could be generated automatically.

Beguelin et al. [B1] state that there are several platforms on which HeNCE could have been built, including Linda, but they comment that Network Linda does not support heterogeneous data formats automatically. We are not sure what is meant by "automatically" or what version of Network Linda was being considered, but the current version certainly does support heterogeneous data formats by means of optional conversion through XDR.

12 Piranha

Piranha is an alternative programming environment which is included as part of the Network Linda product. Although the (theoretical) Piranha model is completely general and makes no assumptions about how parallel programs are organized, SCA's implementation uses tuple space for interprocess communication and Piranha processes use *in* and *out* operations to access tuple space.

The central design goal of the Piranha model is the harnessing of idle CPU cycles available on the nodes of a local area network. It is intended to be more flexible or dynamic than standard Linda in the deployment of processors. The number of processes (piranhas) participating in the parallel program execution can expand and contract during the course of the run in response to changing usage levels on individual systems. For example, if a node becomes free after a Piranha program has started, it can still join in the execution. On the other hand, if the load on a participating node should increase during the run, execution of the Piranha program can terminate on it without disturbing the other nodes and without affecting the final program results.

Unlike Linda programs, Piranha programs do not use *eval* operations to initiate new processes. In fact, they do not explicitly start worker processes at all; this function is handled automatically by the Piranha system. Piranha programs are structured around three key routines.

feeder — *which runs on the system where the user initiated execution This routine usually creates and distributes tasks and gathers results.*

piranha — *which acts essentially as a worker. This is the routine run by the processes created by the Piranha system.*

retreat — *which is called whenever a Piranha process must terminate in mid execution because the system on which it is running is required for other purposes.*

While Piranha is not strictly relevant to a comparison of Network Linda and PVM, it has been included in this report as it is potentially a useful environment and one which comes free with Network Linda. It could be used to address some of the complaints from users of workstations who are sitting at the console and who cannot work effectively because of background tasks. In simpler situations writing a Piranha program is very similar to writing a Linda program and, as in Linda, the user is spared the chore of explicit message passing. The conversion of one of our Linda programs was accomplished with very little effort, possibly because the Linda program had a simple master–worker structure.

Linda represents a versatile parallel programming model with complete generality of decomposition into concurrent computational units. Where different types of subcomputation need to be performed, separate processes are designed for each. In Piranha, some of Linda's expressiveness and generality of decomposition is lost because the parallel program consists basically of a master program (feeder) feeding tasks to a dynamic set of identical workers (piranhas). This means that the separate computational functions of distinct Linda processes are combined into one multi–purpose function replicated in all the Piranha processes. The reason for this is sound inasmuch as when a participating processor is withdrawn, the computation it was performing must be able to be undertaken by any other processor. However, there is an attendant loss of power to express more complex interactions in the intuitively natural style supported by Linda and some ingenuity may be required to compensate for this loss of eloquence.

13 Infelicities in running Network Linda and PVM

As with most software, both of these systems can crash, can leave programs hanging and can also leave defunct processes lying around. Sometimes this is due to the user not making provision for error conditions, but not always.

When PVM runs, it creates a file */tmp/pvmd.<uid>* (where *<uid>* is the user's UNIX uid) on all the machines in the virtual machine and a file */tmp/pvml.<uid>* on machines where an application is initiated. (The file */tmp/pvmd.<uid>* contains a socket address that is used by PVM and the file *pvml.<uid>* is a log file, and is used for PVM messages). After using PVM, it is necessary to remove these files from every machine in the virtual machine, otherwise errors may occur when PVM is next used on these machines. It is also advisable to kill any instances of the PVM daemon *pvmd* that are still around, as sometimes happens.

Similarly, when a Network Linda program does not terminate gracefully, it is sometimes necessary to kill processes in order that the next application can run smoothly.

Another problem with Network Linda can occur when approaching (but not yet exceeding) ones file space quota. A program can hang or give unexpected error messages, until file space utilisation has dropped sufficiently below quota.

During the execution of a Network Linda application the system assumes that all the processes are essential. "Keepalive" messages are regularly transmitted between the processes and if a process stops responding the entire run is terminated. Linda lends itself admirably to job farming. However, in this type of application, the failure of one job often has no bearing on the others and ideally they should not be affected. However, a keepalive failure results in the abortion of the entire run. It is possible to disable the keepalive checks but while we believe that this reduces the chance of total abortion we are not certain that this eliminates that possibility completely. The manner in which the tuple space is distributed over all the workstations may mean that failure of one machine could have implications beyond its own tasks because of the loss of part of the shared tuple space. Unfortunately, running Network Linda on a set of shared public workstations means that the possibility of a machine problem necessitating a reboot or other drastic remedial action is not remote.

PVM has a facility for allowing the loss of a processor to be detected and since messages are sent "point to point", there is no problem analogous to the loss of part of tuple space.

14 Summary of Common Characteristics of Network Linda and PVM

Some of the things that Network Linda and PVM have in common are the following:

- They both enable a heterogeneous collection of computers to work together and appear as one large parallel computing resource.
- They are both layered above the native operating system.
- They both exploit distributed concurrent processing.
- Both can deal with user login names being different on different machines and with files being in different places on different machines.
- They both use TCP/IP protocols, although this may change in future.
- They are both suitable for coarse–grained parallelism.
- They are flexible and general–purpose. With few exceptions (if any) any application that could be programmed on one system could be programmed on the other, although some operations are easier on one or other of the systems.
- They both support both C and FORTRAN versions and, in both, the FORTRAN version is built on top of the C version.
- They exhibit comparable performance and can give impressive speed–ups.

They have both been used on large programs including “Grand Challenge” ones.

They are both evolving products and are continually being enhanced.

They are both available on a number of machines and the range of machines supported is continually being increased.

UNIX style manual pages are available for both.

Both have responsive and helpful support teams. (lsupport@sca.com for Network Linda and pvm@msr.epm.ornl.gov for PVM)

15 Summary of some of the main differences between Network Linda and PVM

Some of the differences between Network Linda and PVM are listed below.

Network Linda	PVM
A commercial product.	Free. Source code available via ftp.
Available for a reasonable range of architectures.	Available for nearly thirty architectures.
Is not currently available for any distributed memory machines.	Is available for some distributed memory machines.
Documentation is in step with the released version.	Documentation may lag behind the most recent version.
Reference manuals are not available on–line and additional copies have to be bought.	Documentation is free and is available on–line via anonymous ftp.
Requires a UNIX operating system.	Mainly UNIX OS but some recent releases have not required UNIX.
Requires a Linda compiler as well as libraries.	Does not need a special compiler.
Uses “tuple space” for IPC.	Uses message passing for IPC.
Simple, rich programming semantics.	More primitive, mechanistic semantics involving explicit message passing.
Limited to running on the number of nodes that are licensed.	No limit imposed on the number of nodes.
Network Linda determines where a process runs.	The programmer specifies where a process runs.
Only one program is involved (This may be a different program on different machines.)	Several programs are involved. These contain PVM library calls.

The newsgroup comp.parallel.linda is not currently active.	The newsgroup comp.parallel.pvm is very active.
Extremely good debugging support including visual support.	Debugging support exists but it is hard work. (XPVM may rectify this.)
There is a start–up overhead for the Network Linda system on each execution.	Any system start–up overhead takes place when the virtual machine is started.
As yet there is no environment which creates a Linda program from a synchronisation specification.	HeNCE will create, build, run and trace a PVM program from sequential code and a graphical synchronisation specification.
The Piranha system which enables programs to run only when there is spare capacity is included as part of the Network Linda product.	There is no equivalent of the Piranha system.

16 Conclusions

Both Network Linda and PVM are effective ways of using a network of computers, working in parallel, to solve a problem. They perform similarly with the same number of machines although PVM seems to be marginally faster.

The main similarities and differences are summarized in the previous two sections. We personally prefer Network Linda because of the elegance of the Linda model, the ease of programming in Linda and the very good development and debugging tools provided by the Code Development System, Tuplescope and the Post Mortem Analyser. However, there are situations which are awkward to handle in Linda because of the atomic and static structure of tuples.

Newcastle University's current Network Linda licence limits its use to 10 nodes. For some large programs many more nodes are needed if a result is to be obtained in a reasonable amount of time. PVM does not limit the number of nodes that can be used. Moreover, it is free and one can build it for as many architectures as one likes without needing to buy additional software or obtain any additional licences. However, from the point of view of the computing community in general, a system which limits people to using specified nodes might well be preferable to one which allows any individual to create processes throughout the whole network, regardless of the current load on each node. The interference unconstrained application of parallel environments can cause is potentially a very serious problem. This is an issue which Piranha starts to address.

PVM gives the user more mechanical control over what is done in parallel programs and may be considered more flexible on this account. However, it is our opinion, borne out by experience, that more effort is needed to write PVM and that this activity is more error–prone. Moreover, since Linda represents a parallel programming model rather than a set of message passing utilities, it is easier to express the logic of concurrency in Linda. Having said that, HeNCE does provide a high–level parallel specification interface and we have yet to determine to what extent this matches the expressive power of Linda.

17 References

[B1] Beguelin, A., Dongarra, J.J., Geist, G.A., Manchek, R., Sunderam, V.S., "Graphical Development Tools for Network–Based Concurrent Supercomputing". Available by anonymous ftp as netlib2.cs.utk.edu:hence/sci92.ps .

[B2] Beguelin, A., "Xab: A Tool for Monitoring PVM Programs". Available by anonymous ftp as netlib2.cs.utk.edu:pvm/xab/xab_hp93.ps.z .

[B3] Beguelin, A., et al. "HeNCE: A User's Guide Version 1.2". Available by anonymous ftp as netlib2.cs.utk.edu:hence/hug.ps .

[C1] Carriero, N., Gelernter, D., "How to Write Parallel Programs: A Guide to the Perplexed", ACM Computing Surveys, September 1989, pp 323–357.

[C2] Carriero, N., Gelernter, D., "How to Write Parallel Programs: A First Course", MIT Press, 1990.

[C3] Carriero, N., Gelernter, D., "Linda and Message Passing: What have we learned?", Yale University, October 1993.

[D1] Dongarra, J.J., Geist, G.A., Manchek, R., Sunderam, V.S., "Integrated PVM Framework Supports Heterogeneous Network Computing" Computers in Physics, 7, 2, 1993 pp 166–175. Also available by anonymous ftp as netlib2.cs.utk.edu:/pvm3/comp–phy (a shar file) .

[D2] Douglas, C.C., Mattson, T.G., Scultz, M.H., "Parallel Programming Systems for Workstation Clusters", Yale University Department of Computer Science Research Report YALE/DCS/TR–975. Available by anonymous ftp as casper.cs.yale.edu:pub/tr975.ps .

[G1] Geist, A. et al., "PVM 3.0 User's Guide and Reference Manual (draft)", February 1993. Until it was superseded by a later version [G2] it was available by anonymous ftp as netlib2.cs.utk.edu:/pvm3/ug.ps .

[G2] Geist, A. et al., "PVM 3 User's Guide and Reference Manual", May 1993. Available by anonymous ftp as netlib2.cs.utk.edu:/pvm3/ug.ps .

{G3] Geist, G.A., Sunderam, V.S., "Experiences with Network Based Concurrent Computing on the PVM System" Concurrency: Practice and Experience, Vol 4 No. 4, June 1992 pp 293–311. A version of this paper with the title "Network Based Concurrent Computing on the PVM System" can be obtained by anonymous ftp as netlib2.cs.utk.edu:/pvm/pvm_experiences. This is a PostScript file. It differs from the CPE paper to some extent.

[M1] Manchek, Robert, "An Introduction to PVM (Parallel Virtual Machine). Describing PVM Version 3.1". Overhead projector slides for a workshop given August 4–5 1993. Available by anonymous ftp from netlib2.cs.utk.edu in /pvm3/workshop.slides.tar.z.uu .

[M2] Matrone, A., Schiano, P., Puoto, V., "LINDA and PVM: A comparison between two environments for parallel programming", Parallel Computing, 1993, pp 949–957.

[M3] Mattson, T.G., Douglas, C.C, Schultz, M.H, “A Comparison of CPS, Linda Posabyl, PVM and TCGMSH: Two Node Communication Times”. An earlier version of the Yale report TR975 [C3] and one that was previously available by anonymous ftp as casper.cs.yale.edu:pub/tr975.ps .

[P1] “Parallel Virtual Machine Quick Reference Guide Release 3.0”, February 15 1993. Until it was superseded, it was available by anonymous ftp as netlib2.cs.utk.edu:/pvm3/refcard.ps .

[P2] “Parallel Virtual Machine Quick Reference Guide Release 3.0”, May 21 1993. Available by anonymous ftp as netlib2.cs.utk.edu:/pvm3/refcard.ps .

[S1] Scientific Computing Associates, “C–Linda User’s Guide & Reference Manual”, Version 2.0 November 1992.

[S2] Scientific Computing Associates, “C–Linda User’s Guide & Reference Manual”, Version 2.1 .

[S3] Scientific Computing Associates, “FORTRAN–Linda Reference Manual”, 1989.

[S4] Scientific Computing Associates, “Installation and Release Notes for Network and CDS Linda Version 2.5, September 1992.

[S5] Scientific Computing Associates, “Installation and Release Notes for Network and CDS Linda Version 2.5.2, April 1993.

[S6] Sunderam, V., “PVM: A Framework for Parallel Distributed Computing”, Concurrency: Practice and Experience, Vol 2, No. 4, Dec 1990. Available by anonymous ftp as mathcs.emory.edu:pub/vss/pvmsystem.ps.Z .

[T1] Turcotte, L.H., “A Survey of Software Environments for Exploiting Networked Computing Resources”, June 1993. Available by anonymous ftp as bulldog.wes.army.mil:/pub/report.ps.Z . This is approximately 150 pages long. The author asks anyone who fetches the paper to send him e–mail (turcotte@wes.army.mil) so that he can notify them when the next release is completed.

Appendix A How to obtain PVM and PVM documentation

If PVM is already installed on the system then there is no need to build it as one copy of the software is sufficient for all users.

By default PVM is installed in $HOME/pvm3

pvm3/lib	pvmd, libraries, scripts
pvm3/include	header files
pvm3/bin	application binaries

Subdirectory ARCH of the pvm3/bin directory (where ARCH corresponds to the architecture of the machine, for example SUN4 or HPPA) is where PVM expects to find executables. The pvm3/bin directory is usually kept private but the pvm3/lib and pvm3/include directories can be linked symbolically to public copies or to those belonging to someone else by executing, for example:

ln –s /usr/local/pvm3/lib $HOME/pvm3/lib
ln –s /usr/local/pvm3/include $HOME/pvm3/include

If PVM is not publicly available then it will have to be built. This requires the source code. The source code also contains some examples so one might want to obtain the source code to look at these if they are not available publicly. There are various ways of acquiring the PVM source code and documentation:

by e–mail
by anonymous ftp
by anonymous rcp
by using xnetlib

The files available are subject to change. A good way to start is to send email to netlib

% mail netlib@ornl.gov Subject: send index from pvm3

and instructions will be mailed back automatically. The index file that is sent includes information about the above four methods and lists the files that are currently available.

As an illustration of their use, the user's guide (currently for PVM version 3.3) is a PostScript file ug.ps which can be obtained as follows using the four methods

by using email to send the following message to netlib@ornl.gov
send ug.ps from pvm3

by anonymous ftp from netlib2.cs.utk.edu,directory pvm, file ug.ps

by anonymous rcp from netlib2.cs.utk.edu by
rcp anon@netlib2.cs.utk.edu:pvm3/ug.ps ug.ps

by using xnetlib in the following way:
Run xnetlib
Click on "Library"
Select pvm3 (by clicking on it)
Scroll to find the line pvm3/ug.ps and click on that item
Click "Download" in the top menu
Click on "Get Files Now" in the window that appears

If the "Download path" entry is not changed the file will be copied across as $HOME/xnlFiles/pvm3/ug.ps Various things will also appear in the directory $HOME/xnlData. These include the file "index" from pvm3.

Note that netlib supports an anonymous ls facility which enables users to find out what files are in the netlib directories without sending e–mail or using anonymous ftp. For example, to find out the contents of the pvm3 directory and the pvm user group directory, pvm3/pvmug, one executes

rsh netlib2.cs.utk.edu –l anon ls –l pvm3
rsh netlib2.cs.utk.edu –l anon ls –l pvm3/pvmug

Note on xnetlib

Xnetlib is a very useful program if one is running X windows. It has a nice interface and gives access to the many useful things that are found in netlib.

The index file obtained in response to sending the message "send index from pvm3" to netlib@ornl.gov suggests that one obtains xnetlib (source) from the same site by sending the message *send xnetlib.shar from xnetlib*. This involves building xnetlib, but there are ready–built binaries in the directory xnetlib of netlib2.cs.utk/edu which can be obtained by anonymous ftp there. The two binaries of most interest to us at Newcastle are (in compressed form)

xnetlib.hp9000.Z
xnetlib.sun4.Z

In addition to the source, the xnetlib.shar file contains documentation such as a manual page for xnetlib and a PostScript version of a Reference card.

Appendix B Summary of the PVM Library Routines

The following gives a brief description of the PVM 3.0 library routines for both the FORTRAN and C versions of PVM. The descriptions have been taken from Appendix A of the "PVM 3.0 User's Guide and Reference Manual", (draft), February 1993 [G1].

FORTRAN	C	Summary
pvmfaddhost	*pvm_addhosts*	Add one or more hosts to the virtual machine.
pvmfbarrier	*pvm_barrier*	Block the calling process until all processes in a group have called it.
pvmfbcast	*pvm_bcast*	Immediately broadcast the data in the active message buffer.
pvmfbufinfo	*pvm_bufinfo*	Return information about the requested message buffer
pvmfconfig	*pvm_config*	Return information about the present virtual machine configuration.
pvmfdelhost	*pvm_delhosts*	Delete one or more hosts from the virtual machine.
pvmfexit	*pvm_exit*	Tell the local pvmd that this process is leaving pvm.

pvmffreebuf	*pvm_freebuf*	Dispose of a message buffer.
pvmfgetinst	*pvm_getinst*	Return the instance number in a group of a pvm process.
pvmfgetrbuf	*pvm_getrbuf*	Return the message buffer identifier for the active receive buffer.
pvmfgetsbuf	*pvm_getsbuf*	Return the message buffer identifier for the active send buffer.
pvmfgettid	*pvm_gettid*	Return the tid of the process identified by a group name and instance number.
pvmfgsize	*pvm_gsize*	Return the number of members presently in the named group.
pvmfinitsend	*pvm_initsend*	Clear default send buffer and specify message encoding.
pvmfjoingroup	*pvm_joingroup*	Enroll the calling process in a named group.
pvmfkill	*pvm_kill*	Terminate a specified pvm process.
pvmflvgroup	*pvm_lvgroup*	Unenroll the calling process from a named group.
pvmfmcast	*pvm_mcast*	Multicast the data in the active message buffer to a set of tasks.
pvmfmkbuf	*pvm_mkbuf*	Create a new message buffer.
pvmfmstat	*pvm_mstat*	Return the status of a host in the virtual machine.
pvmfmytid	*pvm_mytid*	Enroll this process into pvm on its first call and return the tid of the process on every call.
pvmfnotify	*pvm_notif*	Notify a set of tasks about some event.
pvmfnrecv	*pvm_nrecv*	Non–blocking receive.
pvmfpack	*pvm_pk**	Pack the active message buffer with arrays of the prescribed data type (9 variants – byte, cplx, dcplx, double, float, int, long, short and str).
pvmfparent	*pvm_parent*	Return the tid of the process that spawned the calling process.
pvmfperror	*pvm_perror*	Print the error status of the last pvm call.
pvmfpstat	*pvm_pstat*	Return the status of the specified pvm process.
pvmfrecv	*pvm_recv*	Block until a message with specified message tag has arrived from the specified source and place it in a new active receive buffer.

——	*pvm_ecvf*	Redefine the comparison function used to accept messages.
pvmfsend	*pvm_send*	Immediately send the data in the active message buffer.
pvmfsendsig	*pvm_sendsig*	Send a signal to another pvm process.
pvmfserror	*pvm_serror*	Set automatic error message printing on or off for subsequent pvm calls by this process.
pvmfsetrbuf	*pvm_setrbuf*	Switch the active receive buffer and saves the previous buffer.
pvmfsetsbuf	*pvm_setsbuf*	Switch the active send buffer.
pvmfspawn	*pvm_spawn*	Start new pvm process.
pvmftasks	*pvm_tasks*	Return information about the tasks running on the virtual machine.
pvmfunpack	*pvm_upk**	Unpack the active message buffer into arrays of the prescribed data type (9 variants – byte, cplx, dcplx, double, float, int, long, short and str).