Using parallel programming environments on clusters of workstations

Rudnei Dias da Cunha
Computing Laboratory, University of Kent at Canterbury, U.K.
Centro de Processamento de Dados, Universidade Federal do Rio Grande do Sul, Brasil

Tim Hopkins
Computing Laboratory, University of Kent at Canterbury, U.K.

Abstract. We report our experiences using the parallel programming environments, PVM, HeNCE, p4 and TCGMSG and discuss some aspects concerning the performance and software engineering issues.

A brief overview of each environment is given and a number of case studies written using a number of different programming paradigms are presented. Some of the examples presented are simple "building-blocks" which may enhance the performance of parallel applications, others are complete applications.

Keywords Clusters of computers, heterogeneous computing, PVM, HeNCE, p4, TCGMSG

1 Introduction

During the past few years the use of dispersed, networked computers as a pool of processors cooperating on a task has received considerable attention by industry and academia. Such "metacomputers" have been able in some cases to surpass some single processor, high-speed computers, either in terms of cost-performance ratio or indeed in raw execution time. The winner of the 1992 Gordon Bell Prize in the price/performance category, "Statistical mechanics of polymer solutions", was able to reduce the execution time of the application from *3 hours* on a Cray Y-MP to *10 minutes* using a metacomputer composed of 48 IBM RS/6000, 80 Sun SPARC2 and 64 nodes of an Intel iPSC/860 [13].

This is not surprising since in the last years the development of high-performance commodity chips has raised the processing speeds in such a way that today's workstation is the supercomputer of ten years ago. From 1971 to 1992, the clock speed of CMOS processors has increased 1000 times and since 1985 this speed has quadrupled every three years [14]. Increase in performance has also affected network data transmission, memory chip density and disk storage. In 1990 an FDDI network could transmit data at 100 Mbit/s, ten times faster than an Ethernet network of 1980's vintage. A similar trend has affected memory chips where their storage capacity has had a thousandfold increase from 1972 to 1990.

These technological developments favour the metacomputer concept. Indeed, some computer companies like HP and IBM are (re)entering the parallel processing arena with products based on a combination of high-performance commodity processors and high-speed, low-latency networks; Cray Research is preparing to launch a scalable, massively-parallel machine using DEC Alpha processors coupled with vector processing units.

In terms of software, a variety of packages are available which allows the control of the resources in the metacomputer. This report will discuss some of the aspects concerning four parallel programming environments, namely PVM, HeNCE, p4 and TCGMSG. We provide an overview of these packages and then proceed to examine some of the issues involved in running applications written using these packages on a cluster of workstations. We do not intend to provide an extensive analysis of these packages but rather to concentrate on the development of parallel applications and their performance. The reader is referred to [14] for a comprehensive review of the software available for controlling networked resources.

2 Parallel programming environments

The parallel programming environments covered in this report share a number of common characteristics,

- Computation model: an application is a collection of *asynchronous*, concurrent *sequential* processes, interacting through messages exchanged during their execution time.
- *Portability* of source code across different architectures.
- *Transparent* use of different architectures in the same application.
- The possibility of using geographically dispersed machines.
- Seamless support for mixed language (C or Fortran 77) applications.
- *Explicit* and *manual* partitioning and scheduling of an application.
- Support *function-based* distribution of processes.

The machines are regarded as if they were connected via some network and that point-to-point communication between any two nodes is available. For example, a network could be like that in Figure 1 where a number of different machines are depicted.

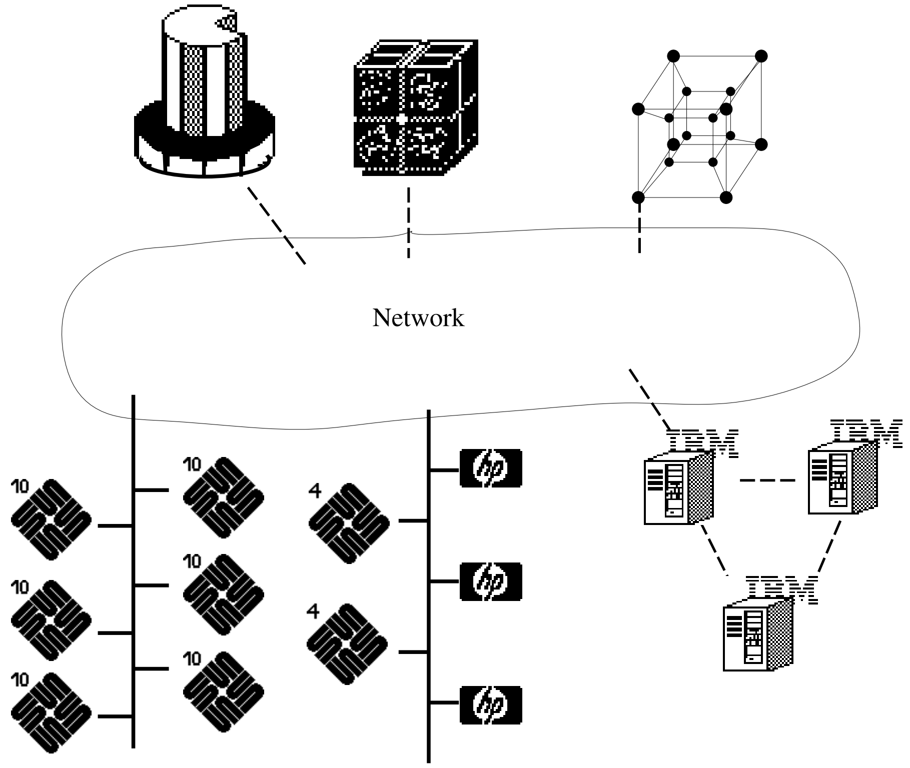
The subsequent sections give some details concerning PVM, p4 and TCGMSG.

2.1 PVM

PVM, or Parallel Virtual Machine, developed by a team at Oak Ridge National Laboratory, University of Tennessee and Emory University [9], is a software package which enables the use of heterogeneous architectures to be used on a single application. PVM-based application can be regarded as a collection of remote processes that interact with each other by exchanging messages during their execution. The current version of PVM is 3.1.5 while an older version, 2.4.2 is still being used. The user is referred to Grant and Skjellum's report on PVM 2.4.1 [10] for an in-depth analysis of the system; some of their comments apply to the current version as well.

PVM is composed of a dæmon, running in each machine used in the application, and a library of functions which are called by the processes in the application. The dæmon is responsible for

Figure 1: Example of a network.



- Initiating remote processes at a request made by a process.
- Transferring messages between application processes.
- Providing information regarding the status of the machines.

PVM also provides a "console" program that allows the user to interactively inquire about the configuration of machines (name of the machine, task id, architecture type, maximum message size, relative speed), add and delete machines, activate and destroy processes, and send signals to processes.

The machines are usually interconnected via an industry-standard network like Ethernet or FDDI[1]; in the case of an Ethernet network PVM uses either the UDP protocol to exchange messages between the remote processes via the dæmons or TCP/IP sockets for point-to-point communication between the remote processes. These two methods of ferrying messages can be used on a single application, e.g. some machines may use the UDP protocol and others may use the point-to-point communication. On multiprocessors like hypercubes or the Intel Paragon XP, PVM uses proprietary system calls to exchange messages across the network connecting the processing elements.

If the user requires socket connections, it must be noted that these are established between the communicating processes the *first* time they exchange messages. This initial transfer thus has the additional cost of setting up the connections.

The process of exchanging messages in PVM consists of two different phases by both the sender and receiver processes

[1]A release of PVM with FDDI support was not publicly available at the time of writing.

Sender 1. *Pack* the data types into the message buffer,
2. *Send* the message.

Receiver 1. *Receive* the message,
2. *Unpack* the data types from the message buffer.

Using this approach, it is possible in PVM to transfer different data types in the same message, by calling as many "pack/unpack" PVM functions as required, for each different data type values.

One of the main advantages of PVM over other systems like p4 and TCGMSG is the capability of a process to initiate and terminate remote processes *dynamically*. The first process to be started (either at the shell prompt or via the PVM console) can call a PVM function which spawns the remote processes. The selection of a specific machine to execute a remote process can be done in three different modes

Transparent A machine from the pool is allocated using an heuristic based on the machine load and rated performance.

Architecture-dependent A node of a specific architecture is selected.

Node-dependent A specific machine is selected (by its Internet name address).

Once a process is initiated it "enrolls" on PVM and either it can start computing or it can initiate other PVM processes (see §3.2 for an example).

The use of heterogeneous machines is handled by PVM without the need of interference by the user. Through the information stored by the dæmons, PVM knows whether XDR conversion must be performed; the user may bypass this if necessary (e.g., for performance reasons on an homogeneous network).

2.2 HeNCE

HeNCE, the Heterogeneous Network Computing Environment, is a graphical tool for the development of parallel programs, developed by the PVM team and others [2]. Applications written either in C or Fortran 77 with HeNCE 1.4 are run under PVM (either 2.4.x or 3.1.x). The main feature of HeNCE is that it allows a parallel application to be written at a higher level than in the other environments. This is achieved by allowing the programmer to graphically design the interrelationships between the different parts of the application, and explicitly assign the execution of such parts to different machines *without* any reference being made to how the data should be transferred between those parts.

A HeNCE application is written by drawing a graph that represents the temporal relationship between different parts of an application. This task is done interactively with the graphical front-end tool, **htool**. The nodes in the graph can be either *compute* nodes or HeNCE *special* nodes

- Compute nodes have two programs associated with them, called *node* and *source*. These programs can be edited and modified using **htool**.

  The node program is written in the HeNCE language (which uses a C-like syntax) and its purpose is to indicate the data dependencies in the graph. Using the HeNCE language the programmer tells HeNCE the type of data (integers, floats, characters), whether the data is created in the node or inherited from other nodes, and if it is input, output or input/output data. The language allows data to be initialised in the declaration.

The source program is a *sequential* code written in either C or Fortran 77 in the form of subroutines or functions.

- The special nodes represent sequential and parallel flow constructs, including *conditional*, *loop*, *fan* and *pipe* nodes.

Directed arcs are placed by the user to connect the nodes in the order that the application requires. Note that these connections, together with any special nodes present in the graph, imply the order of activation of the nodes during the execution of the application.

Figure 2 shows **htool** while an application is being *composed*; the graph of a simple program (to compute vector inner-products) is being shown together with two editing windows containing both the node and source programs (in Fortran 77). Immediately above the graph drawing area are a number of icons representing compute and special nodes, the latter in pairs to identify the beginning and end of each operation (pipe, loop, fan and conditional respectively). Note in the graph the presence of a pair of fan nodes ("fan-out"/"fan-in") enclosing a compute node; this means that the latter will be replicated a number of times, as described in the node program for the first fan node.

When the user tells **htool** to generate the code for the whole application, HeNCE uses the information present in the graph to produce sections of code, called *wrappers*, which call the appropriate PVM routines to transfer the data between the nodes and activate the source program. Once the application is compiled it can be started from **htool** using a *configuration* matrix entered by the user, which assigns the compute nodes to specific machines for execution. To each node the user assigns a value that represents the cost of executing the node subroutine in that machine. This configuration matrix is then used to generate the configuration file read by the PVM local dæmon when it starts.

Once the user requests PVM to be started the application can be executed. The execution is logged on a trace file and this file can be visually analysed, as shown in Figure 3. A number of VCR-like buttons are available to control the display of the trace file (rewind, stop, step-by-step, play) and a timer. The user may request to have windows opened to display icons specific to the machines used (the **Host Map**) and a time-span graph showing the execution of the application (the **Utilization Graph**). A window containing a legend of the status of each node is also available.

Some applications written under HeNCE may suffer in performance. For instance, global accumulation of data, as needed in the computation of vector inner-products, is not performed in recursive-doubling fashion. This operation is usually a bottleneck in an application and it should be replaced by a tuned version where the compute nodes may be started with a fan-out operation but the compute nodes themselves call a routine which calls PVM routines to send/receive data and accumulate the partial values. Using HeNCE 1.4 and PVM 2.4.1 it is possible to use explicit message-passing code for this optimisation.

2.3 p4

The p4 system, or Portable Programs for Parallel Processors, developed at Argonne National Laboratory [3], is a software package which allows the programming of a variety of machines, including both shared- and distributed-memory architectures. Both types of architecture can be used in a single application.

Figure 2: Building an application with HeNCE.

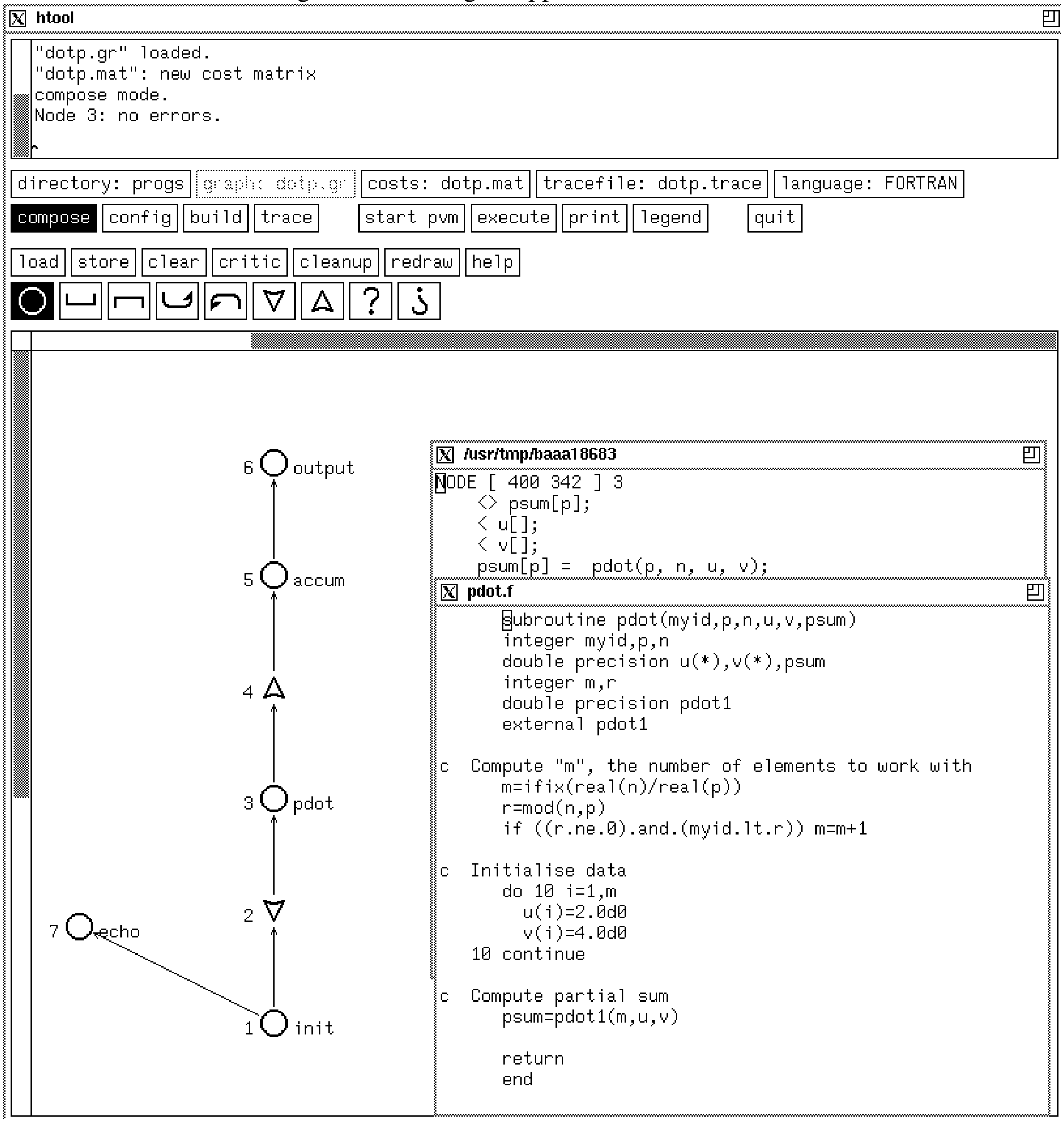
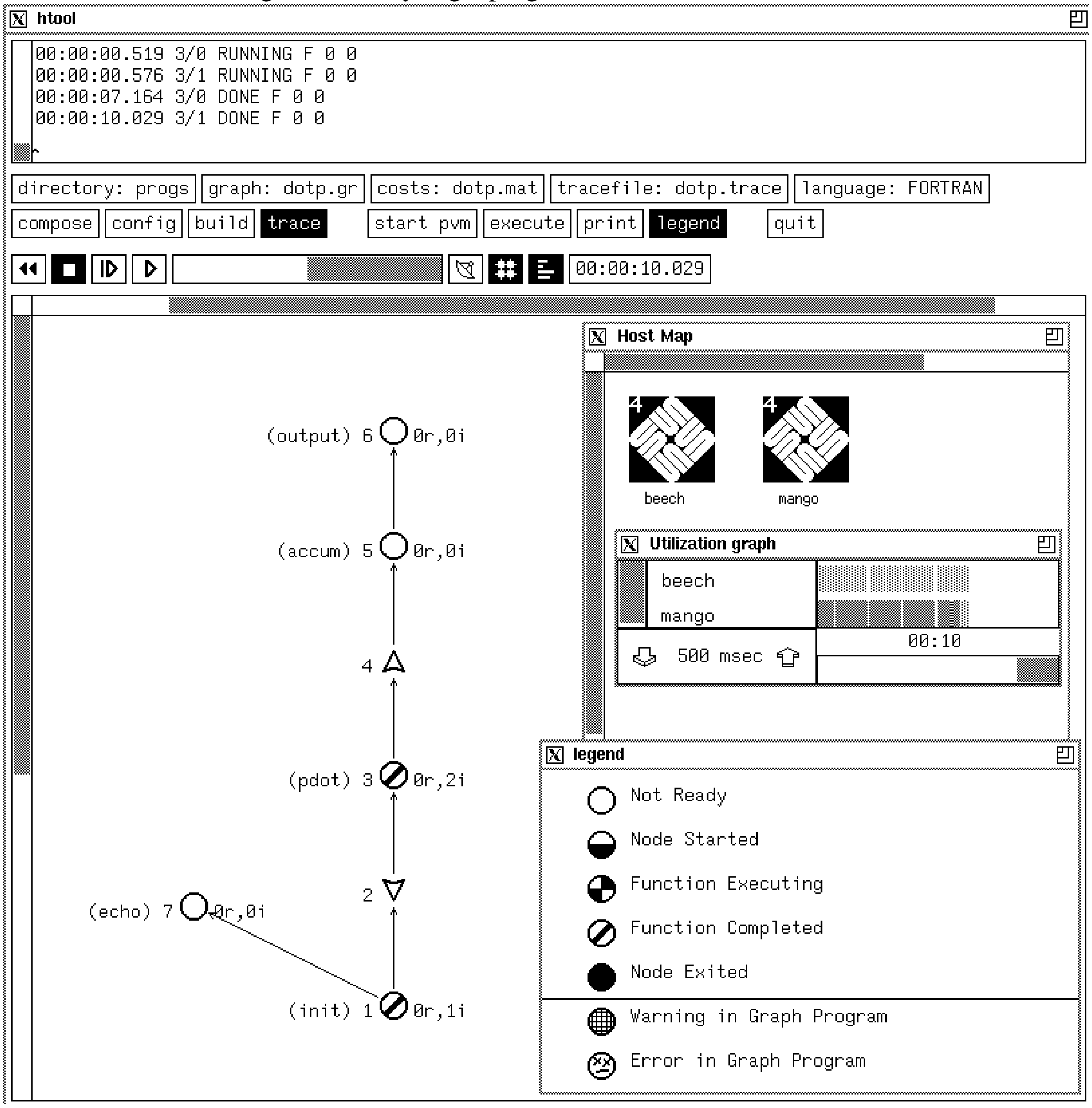


Figure 3: Analysing a program execution with HeNCE.



Messages in p4 are exchanged via TCP sockets on an Ethernet network, either *synchronously* or *asynchronously*. Like PVM, the socket connections are established in run-time when the first message exchange is required between two processes. P4 does not use the packing/unpacking concept of PVM; thus only single data type messages can be exchanged.

Heterogeneous architectures can be used on a distributed-memory application via XDR, but specific send/receive functions must be used for this purpose. The functions provided by p4 are “instrumented”, meaning that logging for debugging or profiling purposes may be obtained although recompilation of the application program(s) may be needed. **Upshot**, a companion program to p4 allows the information stored in the message logging files to be displayed graphically.

Initiation of processes on p4, however, is not dynamic; a file describing which machines will execute which program is read by one of the p4 functions (which must be called by the user) and the remote processes are then initiated via *rsh*. As is well know, remote process execution with *rsh* is slow; to overcome this a dæmon, called *server* is provided only for the purpose of starting up the remote processes[2]. In this case a special file “.p4apps”, residing at the root directory of the user in each of the machines used, must contain the name of the program(s) to be executed[3].

P4 favours the master/slave programming model, since a program must call a “slave” procedure[4]. P4 also provides functions to perform so-called *global operations* over the processes. These functions include maxima and minima, and sums and products and can operate over either vectors or single variables and are quite useful in numerical linear algebra applications.

2.4 TCGMSG

TCGMSG (Theoretical Chemistry Group MeSsaGe-passing system), developed at Battelle Pacific Northwest Laboratory [11], is a system similar to p4. Both shared- and distributed-memory applications are supported. For UNIX machines, TCP sockets are used to ferry the messages between processes in the latter case; in distributed multiprocessors proprietary system calls are used.

Messages are transferred either in synchronous or asynchronous mode; only single data type messages may be transferred. Unlike PVM and p4, however, the socket connections are made between the processes when they are started, whereby each process connects to all the others. In UNIX systems this may cause problems due to the number of open sockets, requiring some tuning of the kernel.

Initiation of processes is not dynamic and is performed via a program called “parallel”, which starts the application programs on the machines via *rsh*, as specified on a configuration file similar to that needed on a p4 application. Use of heterogeneous architectures can be made calling the send/receive functions with specific flags set by the user to obtain XDR conversion.

Automatic logging of messages can be obtained by calling two routines to enable/disable the logging of events – these are stored in a file which may be displayed graphically. Global operations are provided as in p4.

[2]Note that this differs from PVM, where the dæmons may be used as a means of transferring messages.

[3]If the machines share filestore (like NFS or AFS) then just one such file will be needed.

[4]Version 1.3, released at the time of writing, does not have this restriction.

3 Examples

In this section we provide some examples of parallel programs implemented using PVM, HeNCE, p4 and TCGMSG. These examples cover some of the most common operations used in real applications (for example, parallel file access); we also provide examples of SPMD, master-slave and function-based algorithms. An example is also provided in the field of linear algebra, namely a parallel triangular solver. The reader is referred to [6] and [7] for other examples of linear algebra applications.

The tests were carried on the Ethernet-based network of workstations available at the Computing Laboratory at the University of Kent. The machines available comprised Sun SPARC1+, SPARC2 and SPARC10, and HP9000/300.

3.1 Data transmission rates

In this section we present the data transmission rates obtained using PVM 3.1, p4 1.2 and TCGMSG 4.02.

The experiments were carried on two different networks, one being the research network of the Computing Laboratory, which is *shared* by a number of workstations, of which two Sun SPARC2 workstations were used. The other network used is a *dedicated* network with two Sun SPARC1+ workstations. In both cases a network file system was being used.

The data transfer rates were measured by sending 64-bit word messages of different lengths between two workstations using Fortran 77 programs incorporating PVM 3.1, p4 1.2 or TCGMSG 4.02 library routines. The programs used in this experiment are listed in sections A.1–A.3.

The messages were sent in a *round-trip* between the two workstations a hundred times and each round-trip was individually timed. The average time is then used to obtain the data transfer rates. The timing of each round-trip was made calling the UNIX routine `gettimeofday` which returns the elapsed time. The timing routines can be found in section A.1.1.

For the PVM version, the messages were sent using the dæmons with the UDP protocol and point-to-point communication with TCP sockets, with and without XDR conversion. Note that since the networks consisted of homogeneous machines, the conversion will not take place although some overhead is involved when requiring the unnecessary conversion to be done. In the p4 and TCGMSG versions the messages were sent in synchronous and asynchronous mode with and without XDR conversion.

Tables 1–6 show the execution time (in seconds) and the transfer rate (in Mbit/s) obtained for this experiment. We would like to note that our results are similar to those presented by Douglas *et al.* [8]. The following conclusions can be drawn from those results

1. The TCGMSG program shows the best performance, achieving approximately 80% of the nominal Ethernet rate of 10Mbit/s for the largest message length used.

2. The p4 program has a good performance with more than half of the Ethernet bandwidth being achieved for the largest messages.

3. The PVM program has the lowest transfer rate (at most half that of the other programs).

4. The overhead of XDR conversion is a 25%-45% increase in the execution time.

Note that in this experiment we are interested in obtaining the time necessary for the message to arrive at its destination *and* be made available to the application program. For this reason, the timings in the PVM version included both packing and unpacking the messages; this is possibly the main reason for the apparently poor performance of PVM.

We would like to recall that PVM is the only package of those analysed that allows different data types to be transferred in a single message. However, most numerical applications do not require this capability.

3.2 Starting remote processes

Starting a large number of remote processes over a large network may incur substantial overheads. It is easy to see that if a single process is responsible for starting up the remote processes, it becomes a bottleneck which can account for a significant proportion of the overall execution time of the application. We investigated the possibility of using a *scattered* initiation of processes and compared such a strategy to the more usual, naïve method of a single process starting up all the remote processes, which we call a *linear* initiation.

Since the scattered initiation is a dynamic procedure, we used PVM 3.1 in this experiment which consisted of starting up the remote processes in a binary-tree-like form. Each process, as it started, received information from its parent node indicating at which level in the tree the parent process lay and the number of levels required. The remote process then checked to see if it needed to start two more sons. This procedure was repeated until all processes had been started.

A comparison between the scattered and linear initiation procedures was made on a network of Sun workstations. Every time a process was started it selected two different machines on which to start its sons, if any. For a number of 255 processes, the scattered procedure was completed in 15s wall-clock time while the linear procedure required 34s.

A simple analysis of these procedures via an analytical model is sufficient to show that the scattered procedure is superior for a large number of processes. Suppose that $p = 2^h - 1$ processes are required (h denoting the height of the binary-tree) and that T_r is the time required to start up a remote process. Then the total execution time of the scattered and linear initiations may be given respectively by

$$T_S = 2(h-1)T_r$$

$$T_L = (p-1)T_r$$

from which we may conclude that $T_S < T_L$ for $h > 2$.

3.3 Parallel triangular solver

The solution of systems of linear equations is required in many scientific and engineering applications. Triangular systems arise during the solution of linear systems when using direct methods based on *LU* and Cholesky factorisations and for the preconditioning step in association with various iterative methods.

Parallelising the traditional forward- or backward-substitution methods for lower and upper triangular systems respectively requires the use of a *cyclic* or *wrap-around* partitioning of the coefficient matrix among the processors. This is due to the inherently sequential nature of both methods which is overexposed when the coefficient matrix is partitioned in contiguous blocks,

Table 1: PVM 3.1: Asynchronous, point-to-point communication with TCP sockets.

No of	Dedicated network/SPARC1+				Shared network/SPARC2			
64-bit	With XDR		Without XDR		With XDR		Without XDR	
words	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s
1	0.0055	0.02	0.0061	0.02	0.0036	0.04	0.0040	0.03
2	0.0056	0.05	0.0060	0.04	0.0036	0.07	0.0036	0.07
10	0.0059	0.22	0.0066	0.19	0.0041	0.31	0.0038	0.34
20	0.0066	0.39	0.0069	0.37	0.0053	0.49	0.0042	0.61
100	0.0100	1.29	0.0087	1.47	0.0074	1.73	0.0062	2.07
200	0.0154	1.66	0.0133	1.93	0.0121	2.12	0.0087	2.94
1000	0.0615	2.08	0.0412	3.11	0.0880	1.45	0.0567	2.26
2000	0.1326	1.93	0.1012	2.53	0.1500	1.71	0.1174	2.18

Table 2: PVM 3.1: Asynchronous, communication via dæmons.

No of	Dedicated network/SPARC1+				Shared network/SPARC2			
64-bit	With XDR		Without XDR		With XDR		Without XDR	
words	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s
1	0.0155	0.01	0.0136	0.01	0.0116	0.01	0.0197	0.01
2	0.0162	0.02	0.0137	0.02	0.0120	0.02	0.0102	0.03
10	0.0143	0.09	0.0143	0.09	0.0158	0.08	0.0123	0.10
20	0.0153	0.17	0.0150	0.17	0.0131	0.19	0.0149	0.17
100	0.0196	0.65	0.0176	0.73	0.0131	0.98	0.0136	0.94
200	0.0284	0.90	0.0249	1.03	0.0196	1.31	0.0171	1.50
1000	0.0856	1.50	0.0680	1.88	0.0606	2.11	0.0434	2.95
2000	0.1491	1.72	0.1227	2.09	0.1269	2.02	0.0941	2.72

Table 3: P4 1.2: Asynchronous, point-to-point communication with TCP sockets.

No of	Dedicated network/SPARC1+				Shared network/SPARC2			
64-bit	With XDR		Without XDR		With XDR		Without XDR	
words	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s
1	0.0052	0.02	0.0051	0.03	0.0036	0.04	0.0038	0.03
2	0.0057	0.05	0.0053	0.05	0.0038	0.07	0.0044	0.06
10	0.0066	0.19	0.0059	0.22	0.0043	0.30	0.0035	0.36
20	0.0092	0.28	0.0062	0.41	0.0056	0.46	0.0044	0.58
100	0.0113	1.13	0.0094	1.36	0.0070	1.82	0.0046	2.77
200	0.0163	1.57	0.0105	2.43	0.0089	2.87	0.0072	3.55
1000	0.0482	2.66	0.0373	3.44	0.0368	3.48	0.0234	5.47
2000	0.1179	2.17	0.0637	4.02	0.0643	3.98	0.0440	5.81

Table 4: P4 1.2: Synchronous, point-to-point communication with TCP sockets.

No of 64-bit words	Dedicated network/SPARC1+				Shared network/SPARC2			
	With XDR		Without XDR		With XDR		Without XDR	
	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s
1	0.0066	0.02	0.0071	0.02	0.0044	0.03	0.0042	0.03
2	0.0068	0.04	0.0066	0.04	0.0048	0.05	0.0043	0.06
10	0.0080	0.16	0.0069	0.19	0.0063	0.20	0.0046	0.28
20	0.0106	0.24	0.0071	0.36	0.0068	0.37	0.0046	0.56
100	0.0146	0.88	0.0081	1.58	0.0080	1.60	0.0053	2.41
200	0.0197	1.30	0.0112	2.28	0.0117	2.19	0.0080	3.20
1000	0.0454	2.82	0.0336	3.81	0.0313	4.09	0.0239	5.35
2000	0.1153	2.22	0.0654	3.92	0.0709	3.61	0.0457	5.60

Table 5: TCGMSG 4.02: Asynchronous, point-to-point communication with TCP sockets.

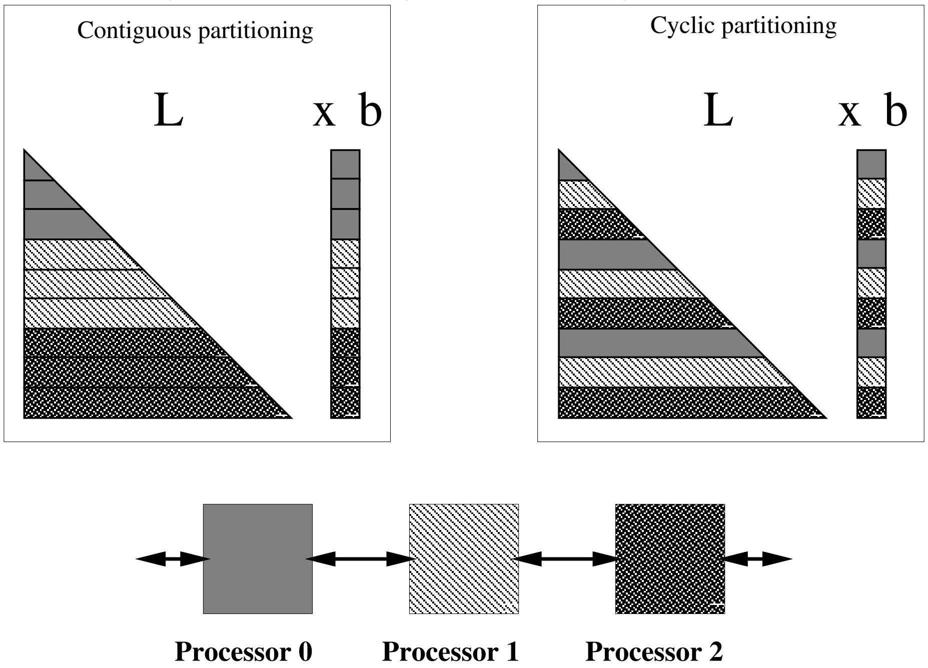
No of 64-bit words	Dedicated network/SPARC1+				Shared network/SPARC2			
	With XDR		Without XDR		With XDR		Without XDR	
	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s
1	0.0049	0.03	0.0047	0.03	0.0031	0.04	0.0036	0.04
2	0.0048	0.05	0.0046	0.06	0.0032	0.08	0.0031	0.08
10	0.0064	0.20	0.0049	0.26	0.0043	0.30	0.0039	0.32
20	0.0066	0.39	0.0050	0.51	0.0107	0.24	0.0042	0.62
100	0.0095	1.35	0.0065	1.96	0.0179	0.72	0.0049	2.61
200	0.0144	1.77	0.0094	2.71	0.0118	2.18	0.0077	3.34
1000	0.0450	2.85	0.0217	5.89	0.0319	4.01	0.0219	5.84
2000	0.0700	3.66	0.0389	6.58	0.0681	3.76	0.0364	7.04

Table 6: TCGMSG 4.02: Synchronous, point-to-point communication with TCP sockets.

No of 64-bit words	Dedicated network/SPARC1+				Shared network/SPARC2			
	With XDR		Without XDR		With XDR		Without XDR	
	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s	Time(s)	Mbit/s
1	0.0052	0.02	0.0046	0.03	0.0045	0.03	0.0032	0.04
2	0.0050	0.05	0.0045	0.06	0.0048	0.05	0.0031	0.08
10	0.0052	0.25	0.0049	0.26	0.0044	0.29	0.0033	0.39
20	0.0059	0.44	0.0051	0.50	0.0042	0.61	0.0033	0.77
100	0.0095	1.35	0.0067	1.90	0.0067	1.92	0.0048	2.67
200	0.0150	1.70	0.0103	2.50	0.0104	2.46	0.0073	3.53
1000	0.0434	2.95	0.0232	5.51	0.0287	4.46	0.0220	5.81
2000	0.0765	3.34	0.0372	6.88	0.0481	5.32	0.0342	7.48

as show in Figure 4. Such a partitioning leads to a situation where the parallel algorithm is slower than the sequential one due to the communication overheads. By cyclically assigning blocks of contiguous rows or columns, the processors are kept busier during the execution of the program and the scalability is increased.

Figure 4: Partitioning of data on triangular solvers.



We have developed an algorithm, called PLTSLES, to solve a lower triangular linear system which was first implemented on a network of transputers using `occam2` (see [5]). The coefficient matrix is partitioned into blocks (*segments*) of contiguous rows of a given size δ; these blocks are then assigned cyclically among the processors. The processors also hold corresponding blocks of the solution vector.

The PLTSLES algorithm uses the forward-substitution method as its basis. For a lower triangular system of n equations

$$Lx = b$$

we initialise x to b and then, for each segment of x the unknowns $(x_k\,,\,x_{k+1}\,,\,\ldots\,,\,x_{k+\delta-1}\,)$ we apply the forward-substitution steps

$$x_k = x_k / L_{k,k} \tag{1}$$

$$x_{k+i} = x_{k+i} - \sum_{j=k}^{i-1} L_{k+i,j} x_j, \quad i = 2, 3, \ldots, \delta - 1 \tag{2}$$

and this segment is said to be *defined*. It may now be broadcast to the other processors which still have $x-$segments to work on and it is also used locally to update the remaining segments using (2). Similarly, as a processor receives a segment of variables, (2) is applied to the segments stored in this processor that have not yet been defined. After all variables needed to update a variable x_k have been received by a processor, (1) and (2) are applied to the variables $x_k\,,\,x_{k+1}\,,\,\ldots\,,\,x_{k+\delta-1}$ and the process is repeated until all processors have all their $x-$segments defined.

This algorithm is similar to the row-wise algorithm developed by Heath and Romine [12] but the messages exchanged between the processors are of a fixed length δ. The main advantage of PLTSLES over the Heath and Romine row-wise algorithm is that it does not have a optimal value of δ after which its scalability degrades substantially. For details, see [5] and [4].

The PLTSLES algorithm was implemented in Fortran 77 and the results in Table 7 were obtained on a network of Sun SPARC 2 workstations using PVM 3.1. The value chosen for the segment size δ affects the performance since a higher communication load results from a small value of δ. As can be seen on Table 7 for $\delta = 2$ and $\delta = 4$ the algorithm does not scale well and may be slower than the sequential forward-substitution method. However as δ and n increase a better performance results.

Table 7: Execution times (in seconds) – parallel triangular solver. Numbers in brackets indicate speed-up.

	Sequen-		δ					
n	tial	p	2	4	8	16	32	64
256	0.06	2	0.28(0.21)	0.10(0.60)	0.06(1.00)	0.05(1.20)	0.04(1.50)	0.02(1.50)
		4	0.18(0.27)	0.10(0.50)	0.05(1.00)	0.02(2.50)	0.01(5.05)	0.01(5.05)
		8	0.15(0.33)	0.06(0.83)	0.02(2.51)	0.03(1.67)	0.01(5.05)	$--$
512	0.24	2	0.58(0.41)	0.39(0.62)	0.19(1.26)	0.10(2.40)	0.10(2.42)	0.10(2.18)
		4	0.52(0.46)	0.25(0.96)	0.14(1.71)	0.09(2.67)	0.09(2.67)	0.06(4.00)
		8	0.33(0.73)	0.14(1.71)	0.08(3.00)	0.07(3.43)	0.06(4.00)	0.04(6.00)
1024	0.84	2	1.37(0.61)	0.87(0.97)	0.46(1.83)	0.42(2.00)	0.43(1.71)	0.38(2.21)
		4	1.02(0.82)	0.57(1.47)	0.41(2.05)	0.26(3.23)	0.20(4.20)	0.19(4.42)
		8	0.78(1.09)	0.44(1.91)	0.17(4.94)	0.09(3.23)	0.15(4.20)	0.08(4.42)
2048	2.94	2	3.98(0.74)	2.46(1.20)	1.73(1.70)	1.85(1.59)	1.68(1.75)	1.83(1.61)
		4	2.60(1.13)	1.36(2.17)	0.84(3.50)	1.04(2.83)	0.93(3.16)	0.86(3.42)
		8	1.49(1.97)	1.11(2.65)	0.61(4.82)	0.54(5.45)	0.53(5.55)	0.34(8.65)

3.4 A master-slave application

Master-slave applications are characterised by having one *master* process and a collection of *slave* processes. The master process is responsible for generating the tasks to be assigned to the slaves and receiving the information produced by the slave processes. Usually the slaves exchange information only with the master process.

As an example application we consider a parallel algorithm to compute the zeros of a complex-valued function $f(z)$. The sequential algorithm makes an exhaustive search over a given initial region $[x\mathit{min}, x\mathit{max}, y\mathit{min}, y\mathit{max}]$. This region is subdivided into n_x and n_y intervals along the real and imaginary axis respectively, resulting in a grid of $n_x n_y$ points.

The search is made by computing $|\, f(z)\, |$ for each point $z = (x_i, y_j)$ on the grid. A point z' is selected as a minimum if $|\, f(z')\, |$ is smaller than the value of $|\, f\, |$ at each of its four neighbouring points in the vertical and horizontal directions. The region containing this point, delimited by $[x_{i-1}, x_{i+1}, y_{i-1}, y_{i+1}]$ is then searched again until the area of a region is smaller than some required tolerance. At each search, several regions may be selected.

This algorithm is inherently parallel as the computation at each grid point is independent of all the others. Since the regions are produced dynamically, allocation of regions to processors cannot be static, which could lead to load-imbalance. A master-slave model is suited to parallelising this algorithm since the master process is responsible for attributing tasks to the slave processes while keeping the application load-balanced (note that other strategies could be used to parallelise the algorithm).

We allowed a certain level of replication among the slave processes, in the sense that the grid points in the interfaces between regions are recomputed by each slave process. To guarantee load-balance, a FIFO queue holding the process identification of the available slaves is maintained by the master process. This queue impedes starvation of tasks by a slave process and guarantees load-balance if there are enough regions to be processed or one or more of the slaves are slower than the others.

The parallel algorithm can be described in terms of its operations by the following

Algorithm 3.1 *Master-slave.*

Master repeat ...
Produces data to the available servers
Receives data back from the servers
Decides if to stop
until no more data is available
Signal slave processes to die

Server repeat ...
Receives data from the master
Sends data back to the master
until death

In algorithm 3.1 different types of messages are required. The three packages PVM, p4 and TCGMSG identify messages by a *user-specified* number and it is important that different numbers are used for different types of messages; this practice avoids execution errors, helps in the debugging and makes the program more readable. The messages we used are detailed in Table 8.

Table 8: Message types for master-slave application.

Message type	Information	Originator	Send type
100	Region: `double r[4]`	Master	Specific to a slave

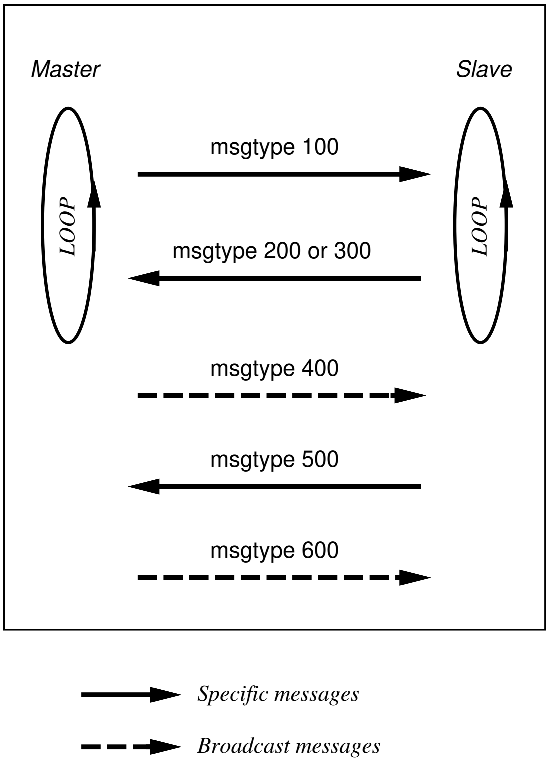
200	New region(s):
	`int no_of_regions`
	`double r[4*no_of_regions]`
		Slave	Specific to master

300	Signals “no regions found”	Slave	Specific to master
400	Gather execution statistics	Master	Broadcast to slaves

500	Execution statistics:
	`int no_of_runs`
	`float exec_time`
		Slave	Specific to master

600	Termination	Master	Broadcast to slaves

Figure 5: Diagram of the master-slave application.

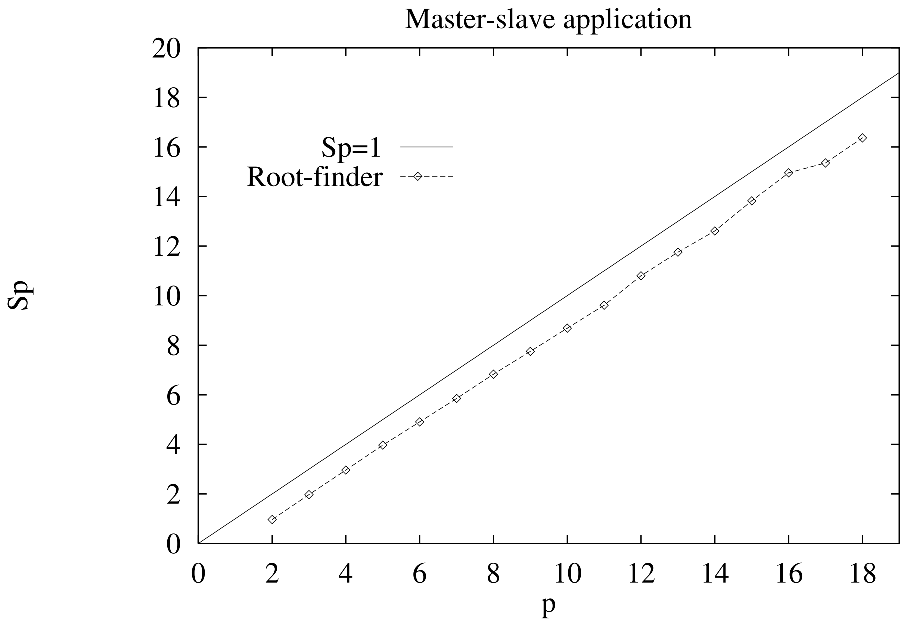


The algorithm was coded in C and was executed under PVM 3.1. We used a heterogeneous network of workstations consisting of

- 6 Sun SPARC 2,
- 2 Sun SPARC 10 and
- 10 HP Apollo 9000 Srs 300

The speed-up results are shown in Figure 6. The speed-up is computed with respect to the sequential algorithm executed on a single workstation. In Figure 6, the values of p indicate 1 master process and $p-1$ slave processes, running on separate workstations. Overall, an efficiency of at least 65% is obtained with this implementation.

Figure 6: Speed-up of the master-slave application.



3.5 An example of a function-based partitioning of an application

Many applications are composed of computational modules which may be executed most efficiently on different types of architectures. For instance, one module may execute most efficiently on a vector processor, while another has an inherent parallelism that matches a distributed-memory machine, and perhaps another is essentially sequential, requiring a very fast scalar processor. Such applications may be partitioned among different machines in terms of the *functions* performed by each section of code.

In this section we provide an example of such an application which is divided into three modules, each running with different hardware/software configuration. This example has the following characteristics

- Two architectures: the implementation uses 2 Sun SPARC2 and an array-processor, an AMT DAP-500 array processor
- Three different languages: C, Fortran 77 and Fortran* (the latter used in the DAP)

The application was the solution of a partial differential equation describing the electrical potential on a circular region. The equation was discretised on a square grid of 32×32 nodes and only those inside the circular region were used; a relaxation technique was then applied to solve the PDE. The values of the electrical potential at the nodes were then displayed in graphical form, depicting the region and associating with each node a different colour according to the value at that point. This application has clearly two distinct phases

1. Solving the PDE
2. Displaying the results

The relaxation technique used in phase 1 makes extensive use of matrix operations and may be executed efficiently on distributed-memory, vector and array-processor architectures. To demonstrate the feasibility of using different architectures in the same application, we decided to use the AMT DAP-500 array-processor with 32×32 processing elements. This machine is linked via a high-speed bus to a Sun SPARC2 workstation which acts as a host to the DAP; this workstation is linked through an Ethernet to the UKC network.

Any DAP application consists of two separate codes: one executes in the host machine and is mainly responsible for I/O operations (from the host to the DAP and back) and starting up the code that runs on the DAP. In our application, the host code was written in Fortran 77 and the DAP code in Fortran*[5], a superset of Fortran 77 with data-parallel extensions similar to those found in Fortran 90. Note that only the host code used PVM to communicate with the display program – it received data from the display program (through PVM calls), sent it to the DAP (through DAP calls), activated the DAP program, received the computed values from the DAP program and sent it back to the display program via PVM.

The display program was written in C and made use of a locally developed, X-windows based graphical library. This program, running on a remote workstation, performs a number of tasks

1. Input of parameters needed for the relaxation technique running on the DAP,
2. Activation of the host program on the DAP front-end,

[5]This program was adapted from an example found in [1].

3. Sending data to the host program and waiting to receive the electrical potential values computed by the DAP, and sent by the host program,

4. Displaying the results by mapping then via a colour look-up table and displaying them in the screen.

Figure 7 shows the relationship between the three programs used in this application and Figure 8 shows the output of the display program.

3.5.1 Using HeNCE

This example was also implemented using HeNCE, after an original implementation had been achieved. Our purpose was to investigate the work necessary to convert an already existing application with explicit message-passing to the HeNCE environment.

As shown in §A.5, there is a single portion of the code in the function **main** which exchanges messages with the DAP host program. The whole program was broken into three separate parts: an *initialisation* module to set parameters for the relaxation technique (in C), a *compute* module (in Fortran 77 and Fortran*) and finally the *display* module (in C). Both the initialisation and the display modules are executed on a Sun SPARC 2 workstation and the compute module is necessarily run on the DAP (host and DAP proper).

These modifications were fairly easy to make, and no modifications were necessary on the DAP code. However, building the application was not straightforward; the use of different languages in the same application is not well supported in HeNCE. In this case, this was circumvented by running **htool** on both workstations, using the *same* graph for the application: on one we built the C modules and on the other the Fortran 77 module. The codes were then installed in the appropriate directories and the application was started from the remote workstation. Figure 9 illustrates **htool** in compose mode with several editing windows open showing the node and source programs.

Figure 7: Diagram of the function-based application.

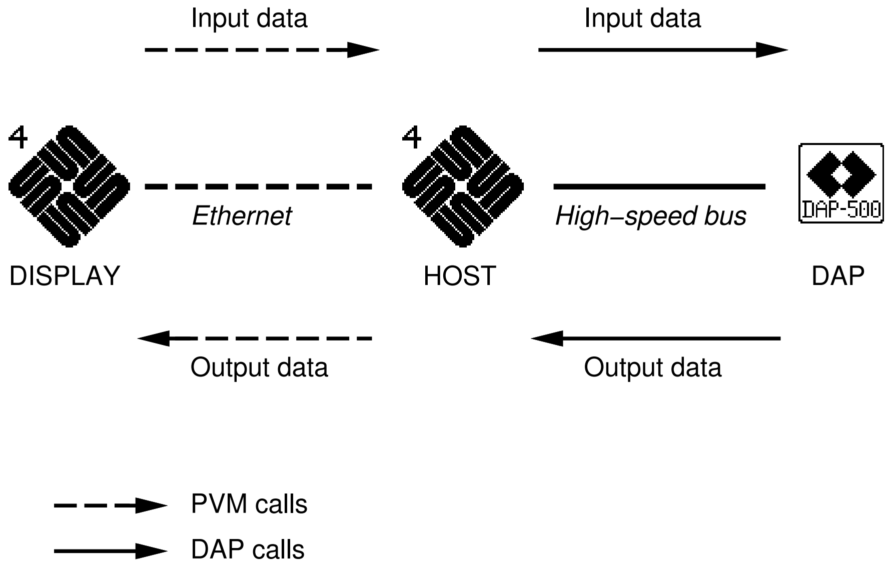


Figure 8: Output of the function-based application.

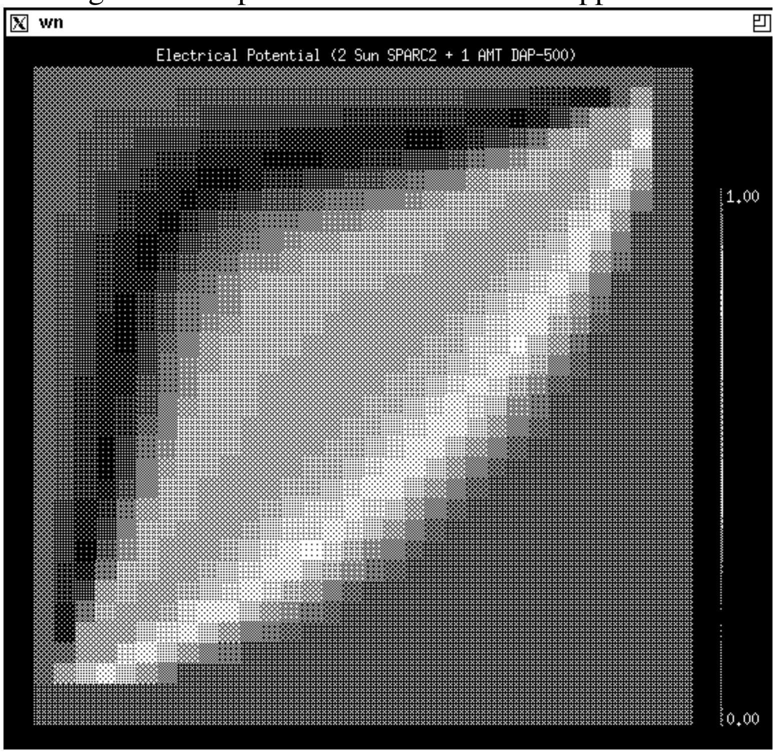
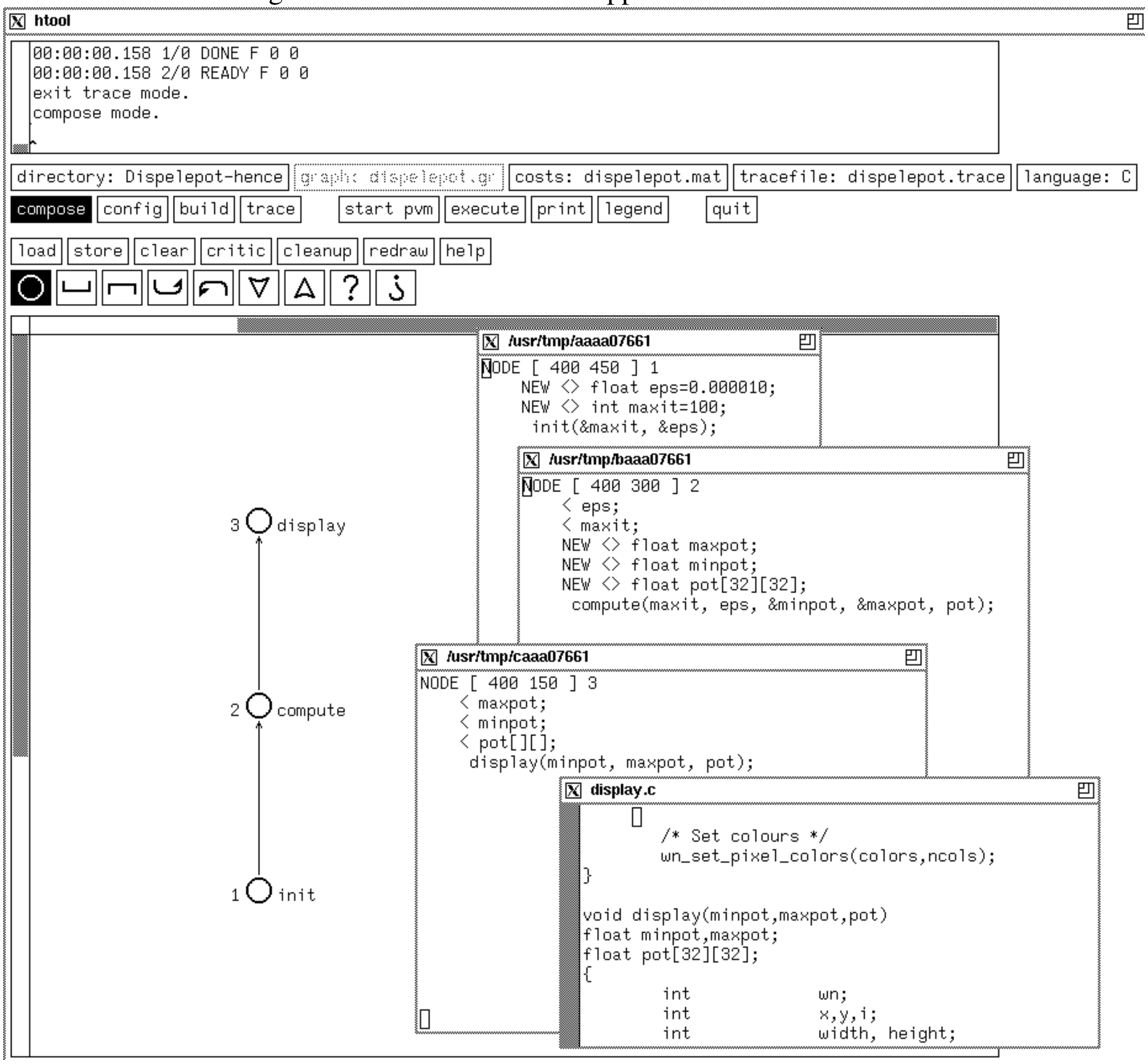


Figure 9: The function-based application under HeNCE.



3.6 *Parallel file access*

Many applications have to read a file to gain access to input data. In a parallel application it is often necessary that simultaneous read accesses to the file are allowed to minimise the start-up time of the application. If this is not possible, then an additional overhead is incurred, that of broadcasting the data from the processing node that has the capability of file access to the other nodes cooperating in the task.

In our case, all the workstations on our local research network share the filestore system using NFS. This characteristic allows for a number of processes to make requests to the NFS file-server to provide the I/O operations on files. It is also possible to have many processes simultaneously reading the same file with some degree of concurrency since NFS servers work asynchronously, buffering the requests made to them. There is a limit to the number of processes that can be satisfactorily serviced by the server dictated, among other reasons, by the number of NFS dæmons running.

We tested the capability of simultaneously reading files using a Fortran 77 program which starts a number of similar processes using PVM. Each process reads a number of records from a file containing 8000 records, the number of records read being a function of the number of processes. The results shown in Table 9 indicate that as we increase the number of processes the scalability decreases, although this performance can be improved by tuning the NFS installation.

As an implementation note, we stress that a PVM 3.1 program reading files will look in the directory where the application program resides (usually `pvm3/bin/ARCH`) unless a specific path to the file is used. The program is listed in §A.6.

Table 9: Execution times – parallel file reading.

p	Records read	Time(s)	S_p
1	8000	1.64	–
2	4000	0.82	2.00
4	2000	0.46	3.57
8	1000	0.30	5.45
16	500	0.19	8.63

4 Summary

In this paper we have presented a number of different applications and their implementations on a number of public-domain parallel programming environments.

Three of the environments, PVM, p4 and TCGMSG, can be used to develop parallel applications across a number of different architectures. In terms of raw data transfer rates, our experiments show that TCGMSG offers the best performance, followed by p4 and PVM. However, this factor may not be sufficient to select one of these environments since they present many different capabilities. PVM allows dynamic initiation of processes while p4 and TCGMSG offer more support for numerical linear algebra applications by providing global accumulation routines. p4 also supports shared-memory architectures which can be used together with distributed-memory architectures in the same application.

Developing applications using these three environments is at a low-level, whereby the programmer explicitly calls routines to send/receive messages. HeNCE is a step forward in the development of parallel applications, freeing the application designer from the details of message-passing. Parallel applications are easily written under HeNCE allowing the re-use of (perhaps old) sections of code, which may be restructured in a single parallel application. Some aspects of performance need to be addressed in HeNCE, particularly when critical sections of code are repeatedly executed. Nonetheless we believe that HeNCE paves the way for the evolution of parallel programming in the future.

References

[1] AMT. *Fortran* User's Manual*, January 1991.

[2] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, R. Wade, J. Plank, and V.S. Sunderam. HeNCE: A user's guide – version 1.2. report, Oak Ridge National Laboratory, December 1992.

[3] R. Butler and E. Lusk. User's guide to the p4 programming system. ANL-92/17, Argonne National Laboratory, October 1992.

[4] R.D. da Cunha. *A Study on Iterative Methods for the Solution of Systems of Linear Equations on Transputer Networks*. PhD thesis, Computing Laboratory, University of Kent at Canterbury, July 1992.

[5] R.D. da Cunha and T.R. Hopkins. The parallel solution of triangular systems of linear equations. In Durand, M. and El Dabaghi, F., editor, *High-Performance Computing II – Proceedings of the Second Symposium on High Performance Computing*, pages 245–256, Amsterdam, October 1991. North-Holland. Also as Report No. 86, Computing Laboratory, University of Kent at Canterbury, U.K.

[6] R.D. da Cunha and T.R. Hopkins. A parallel implementation of the restarted GMRES iterative method for nonsymmetric systems of linear equations. Report No. 7/93, Computing Laboratory, University of Kent at Canterbury, April 1993. Submitted to "Advances in Computational Mathematics".

[7] R.D. da Cunha and T.R. Hopkins. Porting linear algebra subroutines from transputers to clusters of workstations. Report No. 6/93, Computing Laboratory, University of Kent at Canterbury, April 1993. To appear in the proceedings of the "World Transputer Congress" conference, September 20–22, 1993, Aachen.

[8] C.G. Douglas, T.G. Mattson, and M.H. Schultz. Parallel programming systems for workstation clusters. Research report YALEU/DCS/TR-975, Department of Computer Science, Yale University, August 1993.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. PVM 3 user's guide and reference manual. Research Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[10] B.K. Grant and A. Skjellum. The PVM systems: an in-depth analysis and document study – concise edition. Research report, Lawrence Livermore National Laboratory, 1992.

[11] R.J. Harrison. TCGMSG Send/receive subroutines – version 4.02. User's manual, Battelle Pacific Northwest Laboratory, January 1993.

[12] M.T. Heath and C.H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 9:558–588, 1988.

[13] A.H. Karp, K. Miura, and H. Simon. 1992 Gordon Bell Prize winners. *IEEE Computer*, 26(1):77–82, January 1992.

[14] L.H. Turcotte. A survey of software environments for exploiting networked computing resources. Report, Engineering Research Center for Computational Field Simulation, Mississippi State University, June 1993.

A Appendix – Program listings

A.1 Data transmission rate – PVM 3.1 version

```
      program roundtrip

      include 'fpvm3.h'

      parameter (nn=50000)
      parameter (nn1=8)

      integer mytid,numt,info,parenttid,slavetid,mode,advise
      character*8 mach
      character*12 task

c Constants for 'advise' and 'mode' are defined in 'fpvm3.h'
c     parameter (advise = PVMDONTROUTE)
      parameter (advise = PVMROUTEDIRECT)
c     parameter (mode = PVMDEFAULT)
      parameter (mode = PVMRAW)
      parameter (mach = 'beech')

      double precision u(nn)
      integer sizes(nn1),reps
      parameter (reps = 100)
      data sizes/1,2,10,20,100,200,1000,2000/
      real avt,rate,mbit,tsec
      integer s0,s1,s,maxt,mint,tott,usec
      external usec

c     Enroll this program in PVM
      call pvmfmytid(mytid)
      if (mytid.lt.0) then
         print *,'failure in pvmfmytid on round-trip'
         stop
      endif

      call pvmfadvise(advise,info)
      if (info.lt.0) print *,'pvmfadvise(',advise,') failed'

      call pvmfparent(parenttid)
      if (parenttid.lt.0) then

c  I am master

c  Initiate round-trip slave
        task='round-trip'
        call pvmfspawn(task,PVMHOST,mach,1,slavetid,numt)
        if (numt.lt.0) then
          print *,'Failure in pvmfspawn'
          stop
        endif

c  Start bandwidth tests
        write(6,200)
  200 format('PVM 3.1: Message bandwidth, round-trip, ',
```

```
     +'f77 version, ',
     +'double-precision data'/
     +'---------------------------------------------------------------')
        if (advise.eq.PVMDONTROUTE) then
          write(6,210)
  210     format('Timings include packing/unpacking of data.'/
     +'Communication via daemons with UDP')
        else if (advise.eq.PVMROUTEDIRECT) then
          write(6,215)
  215     format('Timings include packing/unpacking of data.'/
     +'Communication with TCP sockets')
        endif
        if (mode.eq.0) then
          write(6,220)
  220   format('With XDR encoding (PVMDEFAULT)')
        else if (mode.eq.1) then
          write(6,230)
  230   format('Without XDR encoding (PVMRAW)')
        endif

c Perform some round-trips before actually timing
        msgtype=100
        do 15 j=1,10
          call pvmfinitsend(mode,info)
          call pvmfpack(REAL8,u,1,1,info)
          call pvmfsend(slavetid,msgtype,info)
          call pvmfrecv(slavetid,msgtype,info)
          call pvmfunpack(REAL8,u,1,1,info)
   15   continue

        do 20 i=1,nn1
          write(6,500)sizes(i)
  500     format('Message length= ',i7,' double-precision words')
          maxt=-1
          mint=999999
          tott=0
          do 30 j=1,reps
            s0=usec()
            call pvmfinitsend(mode,info)
            call pvmfpack(REAL8,u,sizes(i),1,info)
            call pvmfsend(slavetid,msgtype,info)
            call pvmfrecv(slavetid,msgtype,info)
            call pvmfunpack(REAL8,u,sizes(i),1,info)
            s1=usec()
            s=s1-s0
            mint=min(mint,s)
            maxt=max(maxt,s)
            tott=tott+s
   30     continue
          avt=real(tott)/real(reps)
          mbit=64*sizes(i)/1e6
          tsec=avt/2e6
          rate=mbit/tsec
          write(6,1000)avt,mint,maxt,rate
 1000     format('Average=',f15.8,' us, Min =',i6,
     +' us, Max=',i6,' us, Rate=',f15.8,' Mbit/s'/)
```

```
   20   continue

      else

c  I am slave

c Perform some round-trips before actually timing
        msgtype=100
        do 35 j=1,10
          call pvmfrecv(parenttid,msgtype,info)
          call pvmfunpack(REAL8,u,1,1,info)
          call pvmfinitsend(mode,info)
          call pvmfpack(REAL8,u,1,1,info)
          call pvmfsend(parenttid,msgtype,info)
   35   continue

        do 40 i=1,nn1

          do 50 j=1,reps
            call pvmfrecv(parenttid,msgtype,info)
            call pvmfunpack(REAL8,u,sizes(i),1,info)
            call pvmfinitsend(mode,info)
            call pvmfpack(REAL8,u,sizes(i),1,info)
            call pvmfsend(parenttid,msgtype,info)
   50     continue

   40   continue

      endif

c     program finished leave PVM before exiting
      call pvmfexit(info)

      stop
      end
```

A.1.1 The "usec" routines

```
#include <sys/time.h>
#include <time.h>

int usec()
{
        struct timeval  tv;

        gettimeofday(&tv,(struct timezone*)0);
        return tv.tv_sec*1000000+tv.tv_usec;
}

int usec_()
{
        return usec();
}
```

A.2 Data transmission rate – p4 1.2 version

```
      program roundtrip

      include 'p4f.h'

      call p4init()
      if (p4myid().eq.0) then
          call p4crpg()
      endif
      call fslave()
      call p4cleanup()

      stop
      end


      subroutine fslave()

      integer lenint,lenreal,lendble,any
      parameter (lenint=4,lenreal=4,lendble=8,any=-1)

      integer myid,nprocs,msgtype,msglen,reclen,to,from

      include 'p4f.h'

      integer p4myid,p4ntotids
      external p4myid,p4ntotids

      external p4send,p4recv

      parameter (nn=50000)
      parameter (nn1=8)

      double precision u(nn)
      integer sizes(nn1),reps
      parameter (reps = 100)
      data sizes/1,2,10,20,100,200,1000,2000/
      real avt,rate,mbit,tsec
      integer s0,s1,s,maxt,mint,tott,usec
      external usec

c Start

      myid=p4myid()
      nprocs=p4ntotids()

c Start bandwidth tests

      if (myid.eq.0) then
        write(6,200)
  200 format(//'Message bandwidth, round-trip ',
     +'f77/p4 version, ',
     +'double-precision data'/
     +'-----------------------------------------------------')
        write(6,210)
c 210 format('Asynchronous messages.'/'XDR conversion.')
```

```
c 210 format('Asynchronous messages.'/'No XDR conversion.')
c 210 format('Synchronous messages.'/'XDR conversion.')
  210 format('Synchronous messages.'/'No XDR conversion.')

        to=1
        from=1

c Perform some round-trips before timing
        msgtype=100
        msglen=lendble
        do 15 j=1,reps
          call p4send(msgtype,to,u,msglen,info)
          call p4recv(msgtype,from,u,msglen,reclen,info)
   15   continue

        do 20 i=1,nn1
          write(6,500)sizes(i)
  500     format('Message length= ',i7,' double-precision words')
          msglen=sizes(i)*lendble
          maxt=-1
          mint=999999
          tott=0
          do 30 j=1,reps
            s0=usec()
c  Asynchronous/With XDR
c           call p4sendx(msgtype,to,u,msglen,P4DBL,info)
c  Asynchronous/No XDR
c           call p4send(msgtype,to,u,msglen,info)
c  Synchronous/With XDR
c           call p4sendrx(msgtype,to,u,msglen,P4DBL,info)
c  Synchronous/No XDR
            call p4sendr(msgtype,to,u,msglen,info)

            call p4recv(msgtype,from,u,msglen,reclen,info)
            s1=usec()
            s=s1-s0
            mint=min(mint,s)
            maxt=max(maxt,s)
            tott=tott+s
   30     continue
          avt=real(tott)/real(reps)
          mbit=64*sizes(i)/1e6
          tsec=avt/2e6
          rate=mbit/tsec
          write(6,1000)avt,mint,maxt,rate
 1000     format('Average=',f15.8,' us, Min =',i6,
     +' us, Max=',i6,' us, Rate=',f15.8,' Mbit/s'/)

   20   continue
      else

c  I am slave
        to=0
        from=0

c Perform some round-trips before timing
```

```
      msgtype=100
      msglen=lendble
      do 35 j=1,reps
        call p4recv(msgtype,from,u,msglen,reclen,info)
        call p4send(msgtype,to,u,msglen,info)
 35   continue

      do 40 i=1,nn1
        msglen=sizes(i)*lendble
        do 50 j=1,reps
          call p4recv(msgtype,from,u,msglen,reclen,info)

c  Asynchronous/With XDR
c         call p4sendx(msgtype,to,u,msglen,P4DBL,info)
c  Asynchronous/No XDR
c         call p4send(msgtype,to,u,msglen,info)
c  Synchronous/With XDR
c         call p4sendrx(msgtype,to,u,msglen,P4DBL,info)
c  Synchronous/No XDR
          call p4sendr(msgtype,to,u,msglen,info)
 50     continue

 40   continue

      endif

      return
      end
```

A.3 Data transmission rate – TCGMSG version

```
      program roundtrip

      integer lenint,lenreal,lendble,any
      parameter (lenint=4,lenreal=4,lendble=8,any=-1)
      parameter (idbgon=1,idbgoff=0)

      integer myid,nprocs,msgtype,msglen,reclen,sync,
     +to,from,recfrom

      include 'msgtypesf.h'

      integer nodeid, nnodes
      external nodeid, nnodes

      external snd,rcv

      parameter (nn=10000)
      parameter (nn1=8)

      double precision u(nn)
      integer sizes(nn1),reps
      parameter (reps = 100)
      data sizes/1,2,10,20,100,200,1000,2000/
      real avt,rate,mbit,tsec
      integer s0,s1,s,maxt,mint,tott,usec
      external usec

c Start

      call pbeginf

      myid=nodeid()
      nprocs=nnodes()

c Start bandwidth tests

c  Asynchronous (sync=0)/synchronous (sync=1)
c     sync=0
      sync=1

      if (myid.eq.0) then
        write(6,200)
  200 format('Message bandwidth, round-trip, ',
     +'f77/tcgmsg version, ',
     +'double-precision data'/
     +'--------------------------------------------------------')
        write(6,210)
c 210 format('Asynchronous messages.'/'XDR conversion.')
c 210 format('Asynchronous messages.'/'No XDR conversion.')
c 210 format('Synchronous messages.'/'XDR conversion.')
  210 format('Synchronous messages.'/'No XDR conversion.')

        to=1
        from=1
```

```
c Perform some round-trips before timing
        msgtype=100
        msglen=lendble
        do 15 j=1,10
          call snd(msgtype,u,msglen,to,sync)
          call rcv(msgtype,u,msglen,reclen,from,recfrom,sync)
   15   continue

        do 20 i=1,nn1
          write(6,500)sizes(i)
  500 format('Message length= ',i7,' double-precision words')
          msglen=sizes(i)*lendble
          maxt=-1
          mint=999999
          tott=0
          do 30 j=1,reps

c  Round-trip messages
            s0=usec()
c  With XDR
c           call snd(msgtype+MSGDBL,u,msglen,to,sync)
c           call rcv(msgtype+MSGDBL,u,msglen,reclen,from,recfrom,sync)
c  No XDR
            call snd(msgtype,u,msglen,to,sync)
            call rcv(msgtype,u,msglen,reclen,from,recfrom,sync)
            s1=usec()
            s=s1-s0
            mint=min(mint,s)
            maxt=max(maxt,s)
            tott=tott+s
   30     continue
          avt=real(tott)/real(reps)
          mbit=64*sizes(i)/1e6
          tsec=avt/2e6
          rate=mbit/tsec
          write(6,1000)avt,mint,maxt,rate
 1000     format('Average=',f15.8,' us, Min =',i6,
     +' us, Max=',i6,' us, Rate=',f15.8,' Mbit/s'/)

   20   continue
      else

c  I am slave
        to=0
        from=0

c Perform some round-trips before timing
        msgtype=100
        msglen=lendble
        do 35 j=1,10
          call rcv(msgtype,u,msglen,reclen,from,recfrom,sync)
          call snd(msgtype,u,msglen,to,sync)
   35   continue

        do 40 i=1,nn1
          msglen=sizes(i)*lendble
```

```
          do 50 j=1,reps
c  With XDR
c           call rcv(msgtype+MSGDBL,u,msglen,reclen,from,recfrom,sync)
c           call snd(msgtype+MSGDBL,u,msglen,to,sync)
c  No XDR
            call rcv(msgtype,u,msglen,reclen,from,recfrom,sync)
            call snd(msgtype,u,msglen,to,sync)
   50     continue

   40   continue
      endif

      call pend

      stop
      end
```

A.4 Master-slave application

Master program

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <math.h>
#ifdef SUN4
#include <sys/time.h>
#include <sys/resource.h>
#else
#include <time.h>
#endif

#include "pvm3.h"

#include "zroot.h"

#define MAXNHOSTS 18
#define MAXNSLAVES MAXNHOSTS-1

/* SLAVENAME is the name of the slave process */
#define SLAVENAME "zrootps"

void divide(xmin,xmax,ymin,ymax,r,nr)
double  xmin,xmax,ymin,ymax;
double  r[];
int     *nr;
{
/*  Divides a rectangular region into four new subregions of
    length half the x- and y-lengths of the region */
        double  dx,dy;

        dx=xmax-xmin; dy=ymax-ymin;

        *nr=4;
        r[0]=xmin; r[1]=xmin+dx/2.0;
        r[2]=ymin; r[3]=ymin+dy/2.0;
        r[4]=r[1]; r[5]=xmax;
        r[6]=ymin; r[7]=r[3];
        r[8]=xmin; r[9]=xmin+dx/2.0;
        r[10]=r[3]; r[11]=ymax;
        r[12]=r[1]; r[13]=xmax;
        r[14]=r[3]; r[15]=ymax;
}

void topzscan(nslaves,tids,xmin,xmax,nx,ymin,ymax,ny,tol,z,nz,nruns,info)
int     nslaves;
int     tids[];
double  xmin,xmax,ymin,ymax,tol;
int     nx,ny,*nruns,*info;
double  z[];
int     *nz;
{
        double  q[IQSIZE];
        int     qi,qf,ql;
```

```
int     k,h,l,s;
double  dx,dy,nxmin,nxmax,nymin,nymax,norm;
double  r[IRSIZE];
double  aregion[4];
int     nr;
int     toreceive;

/* Queue for slaves */
int     free[MAXNSLAVES];
int     freei,freef,freel,info1;

/* PVM variables */
int     infopvm,msgtype,bufid,slavetid,bytes;

/* Start */
/* Put all the slaves ids into the queue of
   available slaves*/
qinit(&qi,&qf,&ql);
qinit(&freei,&freef,&freel);
*info=0;

for (k=0;k<nslaves;k++) {
  info1=iqins(tids[k],&freei,&freef,&freel,MAXNSLAVES,free);
  if (info1<0) goto bail;
}

/* Subdivide initial region into four new regions */
divide(xmin,xmax,ymin,ymax,r,&nr);

/* Put new regions into queue */
h=0;
for (k=0;k<nr;k++) {
  *info=qins(r[h],&qi,&qf,&ql,IQSIZE,q);
  if (*info<0) goto bail;
  *info=qins(r[h+1],&qi,&qf,&ql,IQSIZE,q);
  if (*info<0) goto bail;
  *info=qins(r[h+2],&qi,&qf,&ql,IQSIZE,q);
  if (*info<0) goto bail;
  *info=qins(r[h+3],&qi,&qf,&ql,IQSIZE,q);
  if (*info<0) goto bail;
  h=h+4;
}

/* Loop through queue, scanning the regions. Put in queue new
   regions if the area is larger than the tolerance, otherwise
   store in the z array
*/
*nruns=0;
*nz=0;
l=0;
while (ql>0) {
  (*nruns)++;

  /* Send regions to available slaves */
  toreceive=0;
  while ((freel>0)&&(ql>0)) {
```

```
            /* Get a slave id from the queue of available slaves */
            info1=iqdel(&slavetid,&freei,&freef,&freel,MAXNSLAVES,free);

            /* Retrieve new region from queue */
            if (*info<0) goto bail;
            *info=qdel(&aregion[1],&qi,&qf,&ql,IQSIZE,q);
            if (*info<0) goto bail;
            *info=qdel(&aregion[2],&qi,&qf,&ql,IQSIZE,q);
            if (*info<0) goto bail;
            *info=qdel(&aregion[3],&qi,&qf,&ql,IQSIZE,q);
            if (*info<0) goto bail;

            msgtype=100;
#ifdef UNIQUE
            bufid=pvm_initsend(PvmDataRaw);
#else
            bufid=pvm_initsend(PvmDataDefault);
#endif
            infopvm=pvm_pkdouble(aregion,4,1);
            infopvm=pvm_pkint(&nx,1,1);
            infopvm=pvm_pkint(&ny,1,1);
            infopvm=pvm_send(slavetid,msgtype);

            toreceive++;
          }

          for (s=0;s<toreceive;s++) {
            /* Receive message from any slave */
            bufid=pvm_recv(-1,-1);
            if (bufid<0) {
              printf("Error on pvm_recv! Bailing out!\n");
              goto bail;
            }

            infopvm=pvm_bufinfo(bufid,&bytes,&msgtype,&slavetid);
            /* Put slave into queue of available slaves */
            info1=iqins(slavetid,&freei,&freef,&freel,MAXNSLAVES,free);

            switch (msgtype) {
              /* New region */
              case 200: /* Unpack region from message buffer */
                        infopvm=pvm_upkint(&nr,1,1);
                        infopvm=pvm_upkdouble(r,nr*4,1);

                        h=0;
                        for (k=0;k<nr;k++) {
                          /* Put new region into queue if region
                             is larger than tolerance */
                          dx=r[h+1]-r[h];
                          dy=r[h+3]-r[h+2];
                          norm=hypot(dx,dy);
                          if (norm<tol) {
                            if (l==INROOT) {
                            *info=-3;
                            goto bail;
                            }
```

```
                              /* Store into z the midpoint of the region */
                              (*nz)++;
                              z[l]=r[h]+dx/2.0;
                              z[l+1]=r[h+2]+dy/2.0;
                              l=l+2;
                            }
                            else {
                              /* Put new region in the queue */
                              *info=qins(r[h],&qi,&qf,&ql,IQSIZE,q);
                              if (*info<0) goto bail;
                              *info=qins(r[h+1],&qi,&qf,&ql,IQSIZE,q);
                              if (*info<0) goto bail;
                              *info=qins(r[h+2],&qi,&qf,&ql,IQSIZE,q);
                              if (*info<0) goto bail;
                              *info=qins(r[h+3],&qi,&qf,&ql,IQSIZE,q);
                              if (*info<0) goto bail;
                            }
                            h=h+4;
                          }
                          break;
                /* No new region */
                case 300: break;
              }
            }
        }

        if (*nz==0) {
          *info=-5;
        }

bail:   ;
}

int
main(argc,argv)
int argc;
char *argv[];
{
        float   rxmin,rxmax,rymin,rymax,rtol;
        double  xmin,xmax,ymin,ymax,tol;
        int     nx,ny,info;
        double  z[INROOT];
        int     nz;
        int     i,j,nruns,slavenruns;
        /* For timing purposes */
#ifdef SUN4
        struct  rusage ru0,ru;
        long    ts,tu;
#else
        clock_t t0,t1;
#endif
        float   t,slavet;
        /* PVM variables */
        int     infopvm,nslaves,mytid,myid,msgtype,bufid,nhosts,narches,
                slaveid;
        int     tids[MAXNSLAVES];
```

```
        struct hostinfo *hosts = 0;

        /* Get input parameters */

        if (argc<8) {
          printf("  usage: %s xmin xmax nx ymin ymax ny nslaves\n",
            argv[0]);
          exit(-1);
        }
        else {
          xmin=(double)atof(argv[1]);
          xmax=(double)atof(argv[2]);
          nx=atoi(argv[3]);
          ymin=(double)atof(argv[4]);
          ymax=(double)atof(argv[5]);
          ny=atoi(argv[6]);
          nslaves=atoi(argv[7]);
        }
        printf("ZROOT Parallel, nslaves=%d\n\n",nslaves);
        printf("Region:\nxmin=%f xmax=%f nx=%d\n",xmin,xmax,nx);
        printf("ymin=%f ymax=%f ny=%d\n",ymin,ymax,ny);

        /* Enroll in PVM */
        myid=0;
        mytid=pvm_mytid();
        if (mytid<0) {
          printf("Failed to enroll! Bailing out!\n");
          exit(-1);
        }

        /* Set socket connections */
#ifdef SUN4
        infopvm=pvm_advise(PvmRouteDirect);
#else
        infopvm=pvm_advise(PvmDontRoute);
#endif
        if (infopvm<0) {
          printf("Failure on pvm_advise! Bailing out!\n");
          pvm_exit();
          exit(-1);
        }

        /* Acquire information about PVM configuration */
        infopvm=pvm_config(&nhosts,&narches,&hosts);
        if (infopvm<0) {
          printf("Failure on pvm_config! Bailing out!\n");
          pvm_exit();
          exit(-1);
        }
        if (nslaves>(nhosts-1)) {
          printf("Not enough processors available! Bailing out!\n");
          pvm_exit();
          exit(-1);
        }

        /* Initiate nslaves instances of SLAVENAME program */
```

```
        for (i=1;i<=nslaves;i++) {
          infopvm=pvm_spawn(SLAVENAME,(char**)0,PvmTaskHost,
                hosts[i].hi_name,1,&tids[i-1]);
          if ((infopvm<0)||(tids[i-1]<0)) {
            printf("Failure on pvm_spawn while starting slave on %s,
        tids=%d! Bailing out!\n",hosts[i].hi_name,tids[i-1]);
            pvm_exit();
            exit(-1);
          }
        }

        /* Send ids to slaves */
        msgtype=50;
        for (i=0;i<nslaves;i++) {
#ifdef UNIQUE
          bufid=pvm_initsend(PvmDataRaw);
#else
          bufid=pvm_initsend(PvmDataDefault);
#endif
          infopvm=pvm_pkint(&i,1,1);
          infopvm=pvm_send(tids[i],msgtype);
        }

        /* Initiate computation */
        tol=1.0e-10;

#ifdef SUN4
        getrusage(RUSAGE_SELF,&ru0);
#else
        t0=clock();
#endif
        topzscan(nslaves,tids,xmin,xmax,nx,ymin,ymax,ny,tol,
          z,&nz,&nruns,&info);

        /* Get execution time */
#ifdef SUN4
        getrusage(RUSAGE_SELF,&ru);
        tu=(ru.ru_utime.tv_sec-ru0.ru_utime.tv_sec)*1000000+
                ru.ru_utime.tv_usec-ru0.ru_utime.tv_usec;
        ts=(ru.ru_stime.tv_sec-ru0.ru_stime.tv_sec)*1000000+
                ru.ru_stime.tv_usec-ru0.ru_stime.tv_usec;
        t=((float)(tu+ts))/1000000.0;
#else
        t1=clock();
        t=((float)(t1-t0))/CLOCKS_PER_SEC;
#endif
        printf("\nMaster   elapsed-time (s)= %f nruns=%d\n",t,nruns);
        /* Request statistics from slaves */
        msgtype=400;
#ifdef UNIQUE
        bufid=pvm_initsend(PvmDataRaw);
#else
        bufid=pvm_initsend(PvmDataDefault);
#endif
        infopvm=pvm_mcast(tids,nslaves,msgtype);
```

```
        /* Receive execution times from slaves */
        msgtype=500;
        for (i=0;i<nslaves;i++) {
          bufid=pvm_recv(-1,msgtype);
          if (bufid<0) {
            printf("Error on pvm_recv! Bailing out!\n");
            pvm_exit();
            exit(-1);
          }

          infopvm=pvm_upkint(&slaveid,1,1);
          infopvm=pvm_upkfloat(&slavet,1,1);
          infopvm=pvm_upkint(&slavenruns,1,1);
          printf("Slave %2d elapsed-time (s)= %f nruns=%d\n",
                slaveid,slavet,slavenruns);
        }

        /* Broadcast termination message to slaves */
        msgtype=600;
#ifdef UNIQUE
        bufid=pvm_initsend(PvmDataRaw);
#else
        bufid=pvm_initsend(PvmDataDefault);
#endif
        infopvm=pvm_mcast(tids,nslaves,msgtype);

        /* Process is finished, leave PVM */
        pvm_exit();

        switch (info) {
          case  0: printf("Roots:\n");
                   printf("No. of roots found=%d\n",nz);
                   /*
                   j=0;
                   for (i=0;i<nz;i++) {
                     printf("z(%d)= %f %f\n",i,z[j],z[j+1]);
                     j=j+2;
                   }
                   */
                   break;
          case -1: printf("Failure: queue full\n");
                   break;
          case -2: printf("Failure: queue empty\n");
                   break;
          case -3: printf("Failure: exceeded number of roots\n");
                   break;
          case -4: printf("Failure: exceeded number of regions\n");
                   break;
          case -5: printf("Failure: no roots found\n");
                   break;
        }

        exit(0);
}
```

Slave program

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <math.h>
#ifdef SUN4
#include <sys/time.h>
#include <sys/resource.h>
#else
#include <time.h>
#endif

#include "pvm3.h"

#include "zroot.h"
#include "funcdef.h"

#define MAXNHOSTS 18
#define MAXNSLAVES MAXNHOSTS-1

double zfunc(x,y,nroots,rr,ri)
double  x,y;
int     nroots;
double  rr[],ri[];
{
        int     k;
        double  a,b,c,d,u,v;

        a=1.0; b=0.0;
        for (k=0;k<nroots;k++) {
          c=x-rr[k]; d=y-ri[k];
          u=a*c-b*d; v=a*d+b*c;
          a=u; b=v;
        }
        return (hypot(a,b));
}

void surpts(x,y,ix,iy,xmin,ymin,dx,dy,nx0,ny0,nroots,rr,ri,px,py,paf)
double  x,y,xmin,ymin,dx,dy,nx0,ny0;
int     ix,iy;
int     nroots;
double  rr[],ri[];
double  px[],py[];
double  paf[];
{
        double  t;
        double  mask[2];
        int     i;

        mask[0]=-1.0; mask[1]=1.0;

        for (i=0;i<2;i++) {
          t=((double)ix+mask[i]-1.0)/nx0;
          px[i]=xmin+t*dx;
          t=((double)iy+mask[i]-1.0)/ny0;
          py[i]=ymin+t*dy;
```

```
        }

        paf[0]=zfunc(px[0],y,nroots,rr,ri);
        paf[1]=zfunc(px[1],y,nroots,rr,ri);
        paf[2]=zfunc(x,py[0],nroots,rr,ri);
        paf[3]=zfunc(x,py[1],nroots,rr,ri);
}

void zscan(xmin,xmax,nx,ymin,ymax,ny,irsize,nroots,rr,ri,r,nr,info)
double  xmin,xmax,ymin,ymax;
int     nx,ny,irsize;
int     nroots;
double  rr[],ri[];
double  r[];
int     *nr,*info;
{
        int     k,h,iy,ix,valid;
        double  af,it,jt,x,y,nx0,ny0,dx,dy;
        double  px[2],py[2],paf[4];

        *nr=0;
        *info=0;
        h=0;
        dx=xmax-xmin;
        dy=ymax-ymin;
        nx0=(double)max(nx-1,1);
        ny0=(double)max(ny-1,1);

        for (iy=1;iy<=ny;iy++) {
          it=((double)(iy-1))/ny0;
          y=ymin+it*dy;
          for (ix=1;ix<=nx;ix++) {
            jt=((double)(ix-1))/nx0;
            x=xmin+jt*dx;
            af=zfunc(x,y,nroots,rr,ri);

            surpts(x,y,ix,iy,xmin,ymin,dx,dy,nx0,ny0,
              nroots,rr,ri,px,py,paf);

            valid=!(isinf(af)||isnan(af));
            for (k=0;(k<4)&&(valid==1);k++)
                valid=!(isinf(paf[k])||isnan(paf[k]));

            if ((valid==1)&&(af<=paf[0])&&(af<=paf[1])&&
                (af<=paf[2])&&(af<=paf[3])) {
              if (h==irsize) {
                *info=-4;
                goto bail;
              }
              /* Store in r(h...h+3)=(px(1),px(2),py(1),py(2)), h=1,5,...
                                     xmin  xmax  ymin  ymax
              */
              r[h]=px[0];
              r[h+1]=px[1];
              r[h+2]=py[0];
              r[h+3]=py[1];
```

```
              h=h+4;
              (*nr)++;
            }
          }
        }

bail:   ;
}

int
main(argc,argv)
int argc;
char *argv[];
{
        double  nxmin,nxmax,nymin,nymax;
        int     nx,ny;
        int     info;
        double  r[IRSIZE];
        double  aregion[4];
        int     nr;
        int     i,j,die,nruns;
        int     nroots;
        double  rr[MAXNR],ri[MAXNR];
        /* For timing purposes */
#ifdef SUN4
        struct  rusage ru0,ru;
        long    ts,tu;
#else
        clock_t t0,t1;
#endif
        float   t;
        /* PVM variables */
        int     infopvm,msgtype,bufid,mastertid,slaveid,mytid,myid,
                recvtid,bytes,nslaves;
        int     tids[MAXNSLAVES];

        /* Enroll in PVM */
        mytid=pvm_mytid();
        if (mytid<0) {
          printf("Failed to enroll! Bailing out!\n");
          pvm_exit();
          exit(-1);
        }

        /* Get master's id */
        mastertid=pvm_parent();
        if (mastertid<0) {
          printf("Failure on pvm_parent! Bailing out!\n");
          pvm_exit();
          exit(-1);
        }

        /* Set socket connections */
#ifdef SUN4
        infopvm=pvm_advise(PvmRouteDirect);
#else
```

```
        infopvm=pvm_advise(PvmDontRoute);
#endif
        if (infopvm<0) {
          printf("Failure on pvm_advise! Bailing out!\n");
          pvm_exit();
          exit(-1);
        }

        /* Receive myid */
        msgtype=50;
        bufid=pvm_recv(mastertid,msgtype);
        if (bufid<0) {
          printf("Error on pvm_recv! Bailing out!\n");
          pvm_exit();
          exit(-1);
        }
        infopvm=pvm_upkint(&myid,1,1);

#ifdef SUN4
        getrusage(RUSAGE_SELF,&ru0);
#else
        t0=clock();
#endif

        /* Define function */
        funcdef(&nroots,rr,ri);

        /* Loop for messages */
        nruns=0;
        die=0;
        while (die!=1) {
          nruns++;
          bufid=pvm_recv(mastertid,-1);
          if (bufid<0) {
            printf("Error on pvm_recv! Bailing out!\n");
            pvm_exit();
            exit(-1);
          }
          infopvm=pvm_bufinfo(bufid,&bytes,&msgtype,&recvtid);

          switch (msgtype) {
            /* New region */
            case 100: /* Unpack region from message buffer */
                      infopvm=pvm_upkdouble(aregion,4,1);
                      infopvm=pvm_upkint(&nx,1,1);
                      infopvm=pvm_upkint(&ny,1,1);

                      /* Scan new region */
                      zscan(aregion[0],aregion[1],nx,
                        aregion[2],aregion[3],ny,
                        IRSIZE,nroots,rr,ri,r,&nr,&info);
                      if (info<0) {
                        printf("Error on zscan, info=%d !
                          Bailing out!\n",info);
                        pvm_exit;
                        exit(-1);
```

```
                    }

                    if (nr!=0) {
                      /* New region has been created */
                      msgtype=200;
#ifdef UNIQUE
                      bufid=pvm_initsend(PvmDataRaw);
#else
                      bufid=pvm_initsend(PvmDataDefault);
#endif
                      infopvm=pvm_pkint(&nr,1,1);
                      infopvm=pvm_pkdouble(r,nr*4,1);
                      infopvm=pvm_send(mastertid,msgtype);
                    }
                    else {
                      /* No new region has been found */
                      msgtype=300;
#ifdef UNIQUE
                      bufid=pvm_initsend(PvmDataRaw);
#else
                      bufid=pvm_initsend(PvmDataDefault);
#endif
                      infopvm=pvm_send(mastertid,msgtype);
                    }
                    break;

          /* My master requested my statistics of usage */
          case 400: /* Get execution time */
#ifdef SUN4
                    getrusage(RUSAGE_SELF,&ru);
                    tu=(ru.ru_utime.tv_sec-ru0.ru_utime.tv_sec)*
                      1000000+ru.ru_utime.tv_usec-
                      ru0.ru_utime.tv_usec;
                    ts=(ru.ru_stime.tv_sec-ru0.ru_stime.tv_sec)*
                      1000000+ru.ru_stime.tv_usec-
                      ru0.ru_stime.tv_usec;
                    t=((float)(tu+ts))/1000000.0;
#else
                    t1=clock();
                    t=((float)(t1-t0))/CLOCKS_PER_SEC;
#endif

                    /* Send time to master */
                    msgtype=500;
#ifdef UNIQUE
                    bufid=pvm_initsend(PvmDataRaw);
#else
                    bufid=pvm_initsend(PvmDataDefault);
#endif
                    infopvm=pvm_pkint(&myid,1,1);
                    infopvm=pvm_pkfloat(&t,1,1);
                    infopvm=pvm_pkint(&nruns,1,1);
                    infopvm=pvm_send(mastertid,msgtype);

                    break;
```

```
        /* My master ordered me to die. */
        case 600: /* Unenroll from PVM */
                  pvm_exit();

                  /* Die swiftly! */
                  die=1;
                  break;
      }
    }

    exit(0);
}
```

A.5 Function-based application

Display program

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <math.h>
#include <local/wn.h>
#include <local/menu3.h>

/* NIC is number of intermediate colours */
#define NIC 38
#define NCOLS 5*NIC+6
#define NCOLS1 NCOLS+2
#define MAX_BRIGHTNESS 65535

#define MYNAME "dispelepot"
#define DAPPROG "elepot"
#define DAPSUN "dapsun.ukc.ac.uk"

#define PSIZE 32

#define WSIZE 520
#define USABLEWSIZE 480
#define BOXSIZE (USABLEWSIZE/PSIZE)
#define STARTMAPPOS 20
#define SLIDESIZE 40
#define SLIDEBOXSIZE 2
#define STARTSLIDEPOSX WSIZE
#define STARTSLIDEPOSY STARTMAPPOS+PSIZE*BOXSIZE-1

#define EVNTMSK EV_BUTTON_DOWN

#include "create_C_colourmap.c"

int
main(argc,argv)
int argc;
char **argv;
{
        int             wn;
        int             x,y,i;
        int             width, height;
        int             nbx,nby;
        event_t         ev;
        colour_t        colourmap[NCOLS1];
        int             colour[PSIZE][PSIZE];
        float           interval[NCOLS];
        float           cinc;
        char            s[20];

        /* PVM variables */
        int             info,dap,msgtype;

        /* Variables for the DAP */
        int             maxit;
```

```
float           eps;
float           pot[PSIZE][PSIZE];
float           minpot,maxpot;

if (enroll(MYNAME)<0) {
  printf("Failure to enroll \"%s\"\n",MYNAME);
  exit(0);
}

if ((dap=initiateM(DAPPROG,DAPSUN))<0) {
  printf("Failure to enroll \"%s\"\n",DAPPROG);
  exit(0);
}

maxit=100;
eps=0.00001;

/* Send maxit and eps to DAP part */
msgtype=100;
initsend();
info=putnint(&maxit,1);
info=putnfloat(&eps,1);
if ((info=vsnd(DAPPROG,dap,msgtype))<0) {
  printf("Failure to send\n");
  info=terminate(DAPPROG,dap);
  exit(0);
}

/* Receive from DAP the electric potential values */
msgtype=200;
info=vrcv(msgtype);
info=getnfloat(&minpot,1);
info=getnfloat(&maxpot,1);
info=getnfloat(pot,PSIZE*PSIZE);

/* DAP application finished, leave PVM */
leave();

/* Compute colour interval */
cinc=(maxpot-minpot)/(NCOLS-1);
for (i=0;i<NCOLS;i++) interval[i]=minpot+i*cinc;

/* Set colours to potential values */
for (y=0;y<PSIZE;y++) {
  for (x=0;x<PSIZE;x++) {
    for (i=0;i<(NCOLS-1);i++)
      if ((pot[y][x]>=interval[i])&&(pot[y][x]<=interval[i+1])) {
        colour[y][x]=i+2;
        break;
      }
  }
}

/* Open window */
wn_suggest_window_size(WSIZE+SLIDESIZE,WSIZE);
if ((wn = wn_open_stdwin()) < 0) {
```

```
                fprintf(stderr,"Can't create the standard window\n");
                exit(1);
        }

        create_C_colourmap(colourmap,NCOLS1);

        /* Display electric potential */
        wn_get_window_size(wn,&width,&height);
        /* clear the window */
        wn_set_area(wn,0,0,width,height,WN_BG);

        /* Write text */
        wn_btext(wn,
          "Electrical Potential (2 Sun SPARC2 + 1 AMT DAP-500)",
                110,15,WN_FG,WN_BG);

        /* Display slide */
        x=STARTSLIDEPOSX;
        y=STARTSLIDEPOSY;
        sprintf(s,"%2.2f",minpot);
        wn_btext(wn,s,x+SLIDEBOXSIZE+3,y,WN_FG,WN_BG);
        for (i=2;i<NCOLS1;i++) {
          wn_set_area(wn,x,y,SLIDEBOXSIZE,SLIDEBOXSIZE,
                colourmap[i].co_pixel);
          y=y-SLIDEBOXSIZE;
        }
        sprintf(s,"%2.2f",maxpot);
        wn_btext(wn,s,x+SLIDEBOXSIZE+3,y+11,WN_FG,WN_BG);

        /* Display map */
        y=STARTMAPPOS;
        for (nby=0;nby<PSIZE;nby++) {
          x=STARTMAPPOS;
          for (nbx=0;nbx<PSIZE;nbx++) {
            wn_set_area(wn,x,y,BOXSIZE,BOXSIZE,
                colourmap[colour[nby][nbx]].co_pixel);
            x=x+BOXSIZE;
          }
          y=y+BOXSIZE;
        }

        /* Wait for mouse click to finish */
        for (;;) {
          wn_next_event(wn,EVNTMSK,ev);
          if ((ev.ev_flags & B_LEFT) && (ev.ev_type & EV_BUTTON_DOWN))
            wn_close_window(wn);
            exit(0);
        }

}
```

Compute program (DAP host)

```
      program elepot

      parameter (isize=32)

      integer psize,niter,maxit,status
      real pot(isize,isize),eps,minpot,maxpot

      common /bl0001/ psize,maxit,eps
      common /bl0002/ niter,status
      common /bl0003/ pot
      common /bl0004/ minpot,maxpot

      integer dapcon
      external dapcon

c PVM declarations

      integer myid,info,msgtype,hostnum
      character*14 myname,hostname

c Enroll on PVM
      hostname="dispelepot\0"
      hostnum=0

      myname="elepot\0"
      call fenroll(myname,myid)
      if (myid.lt.0) then
         print *,'failure in fenroll on ',myname
         stop
      endif

c Receive maxit and eps
      msgtype=100
      call fvrcv(msgtype,info)
      call fgetnint(maxit,1,info)
      call fgetnfloat(eps,1,info)

      psize=isize

c Connect DAP part
      if (dapcon('elepot.dap').ne.0) then
        print *,'Failed to load DAP part ...'
      else
c Send data to DAP
        call dapsen('bl0001',psize,3)

c Initiate DAP part
        call dapent('elepotdap')

c Receive data from DAP
        call daprec('bl0002',niter,2)
        call daprec('bl0003',pot,isize*isize)
        call daprec('bl0004',minpot,2)

c Release DAP
```

```
        call daprel

c Send data back to PVM host
        msgtype=200
        call finitsend()
        call fputnfloat(minpot,1,info)
        call fputnfloat(maxpot,1,info)
        call fputnfloat(pot,isize*isize,info)
        call fvsnd(hostname,hostnum,msgtype,info)

      endif

c Leave PVM
      call fleave()

      stop
      end
```

Compute program (DAP)

```
      entry subroutine elepotdap

      parameter (isize=32)

      integer psize,niter,maxit,status
      real pot(*isize,*isize),eps,minpot,maxpot

      common /bl0001/ psize,maxit,eps
      common /bl0002/ niter,status
      common /bl0003/ pot
      common /bl0004/ minpot,maxpot

c  Converts data from host mode to dap mode
      call convhtod(psize,3)
      call convhtod(pot,1)

c  Performs the actual root-finder algorithm
      call quadrant(pot,psize,eps,niter,maxit,status)

c  Obtains minimum and maximum values of pot
      minpot=minv(pot)
      maxpot=maxv(pot)

c  Converts data back to host mode
      call convdtoh(niter,2)
      call convdtoh(pot,1)
      call convdtoh(minpot,2)

      return
      end


      subroutine quadrant(p,psize,eps,niter,maxit,status)
      integer psize,maxit,niter,status

c Declaration of arrays in Fortran* format
      real p(*psize,*psize),p0(*psize,*psize),eps
      logical curved(*psize,*psize),inside(*psize,*psize),
     1        outside(*psize,*psize)
      integer square(*psize),xory(*psize)

      status=1
c Set boundary conditions
      outside=.false.
      outside(1,)=.true.
      outside(,1)=.true.
      p=0.0

      call index_vec(xory)
      square=xory*xory

      curved=(matc(square,psize)+matr(square,psize)).ge.(psize-1)**2
      p(curved)=1.0

      outside=outside.or.curved
```

```
      inside=.not.outside

c Set initial guess
      p(inside)=0.5

c Solve Laplacian
      do 10 niter=1,maxit
        p0=p
c The expression below is one of the features of
c Fortran*; it indicates to operate with the
c elements one column ahead "p(,+)" and behind "p(,-)"
c and similarly one row above "p(+,)" and below "p(-,)"
c only for those "inside" the region.
        p(inside)=.25*(p(,+)+p(,-)+p(+,)+p(-,))
        if (maxv(abs(p-p0)).lt.eps) return
   10 continue

      status=0

      return
      end
```

A.6 Parallel file access

```
      program readfilep

      include 'fpvm3.h'

      integer i,myid,mytid,numt,info,m,r,firstrec,lastrec,nrecs,
     1mxtsk,mxtsk1
      integer tids(16)
      character*8 arch
      character*12 task

      parameter (nrecs=8000)

      character*80 infil,buffer
      logical fex
      real etime,t0(2),t1(2),t(2),et0,et1,et
      external etime

c  Begin
c  Enroll this program in PVM
      call pvmfmytid(mytid)
      if (mytid.lt.0) then
         print *,'failure in pvmfmytid on readfilep'
         stop
      endif

      call pvmfparent(tids(1))
      if (tids(1).lt.0) then

        print *,'Number of tasks?'
        read *,mxtsk
        mxtsk1=mxtsk-1

c  I am process 0
        myid=0
        tids(myid+1)=mytid

c  Initiate ntasks instances of readfilep program
        task='readfilep'
        arch='*'
        do 10 i=1,mxtsk1
           call pvmfspawn(task,PVMDEFAULT,arch,1,tids(i+1),numt)
           if (numt.lt.0) then
             print *,'Failure in pvmfspawn at task',i
             stop
           endif
   10   continue

c Send number of tasks
        msgtype=8
        call pvmfinitsend(PVMDEFAULT,info)
        call pvmfpack(INTEGER4,mxtsk,1,1,info)
        call pvmfmcast(mxtsk1,tids(2),msgtype,info)

c Broadcast task ids
        msgtype=9
```

```
          call pvmfinitsend(PVMDEFAULT,info)
          call pvmfpack(INTEGER4,tids,mxtsk,1,info)
          call pvmfmcast(mxtsk1,tids(2),msgtype,info)

        else

c I am other process
          msgtype=8
          call pvmfrecv(tids(1),msgtype,info)
          call pvmfunpack(INTEGER4,mxtsk,1,1,info)
          mxtsk1=mxtsk-1

          msgtype=9
          call pvmfrecv(tids(1),msgtype,info)
          call pvmfunpack(INTEGER4,tids,mxtsk,1,info)

          do 15 i=2,mxtsk
            if (mytid.eq.tids(i)) myid=i-1
   15     continue
        endif

c Open the file infil
        infil='test.dat'
        inquire(file=infil,exist=fex)
        if (fex) then
          open(1,file=infil,access='direct',recl=80,status='old')
        else
          write(6,20)
   20     format('File not found! Exiting!')
          goto 999
        end if

c  Set first and last records to read
        m=nint(nrecs/mxtsk)
        r=nrecs-m*mxtsk
        if (r.eq.0) then
c  Simplest case, all processes read "m" records
          firstrec=myid*m+1
          lastrec=firstrec+m-1
        else if (myid.lt.r) then
c  Otherwise, the first "r" processes read "m+1" records
          firstrec=myid*m+myid+1
          lastrec=firstrec+m
        else
c  and the remaining "mxtsk-r" processes read "m" records
          firstrec=myid*m+r+1
          lastrec=firstrec+m-1
        endif

        et0=etime(t0)
        do 30 i=firstrec,lastrec
          read(1,40,rec=i)buffer
   40     format(a80)
   30 continue
        et1=etime(t1)
        et=et1-et0
```

```
      t(1)=t1(1)-t0(1)
      t(2)=t1(2)-t0(2)
      write(6,50)m
   50 format('Read ',i5,' records from file')
      write(6,1000)t(1),t(2),et
 1000 format('User-time(s):   ',f15.8/,'System-time(s): ',f15.8/,
     &'Elapsed-time(s):',f15.8)

      close(1)

  999 continue

c  Program finished, leave PVM before exiting
      call pvmfexit(info)

      stop
      end
```