Overview of VPE: A Visual Environment for Message-Passing

Peter Newton
Jack Dongarra

Computer Science Department
University of Tennessee
Knoxville, TN, 37996-1301

Abstract

VPE is a fully integrated visual heterogeneous parallel programming environment with a message-passing orientation. It is intended to provide a simple human interface to the process of creating message-passing programs. Programmers describe the process structure of a program by drawing a graph in which nodes represent processes and messages flow on arcs between nodes. They then annotate these computation nodes with program text expressed in C or Fortran which contains simple message-passing calls. The VPE environment can then automatically compile, execute, and animate the program, even on heterogeneous targets.

VPE uses PVM as its initial implementation vehicle, but it is designed so as to support targeting other message-passing systems later. It's GUI provides direct user management of a heterogeneous collection of machines to be used as a virtual parallel machine.

1 Introduction

Many existing parallel computing languages and environments are somewhat difficult to use. This fact limits their acceptance among computational scientists, especially when they have little prior experience with parallel programming. Heterogeneous environments greatly exacerbate the difficulties by complicating the programmer's machine model and for more prosaic reasons such as the difficulties involved in managing compiles on multiple machines.

VPE addresses these issues by providing an integrated visually-oriented programming environment which allows users to

1. Express process structure visually, thus simplifying the programming model,
2. Directly manage the hosts in a heterogeneous virtual parallel machine,
3. Automatically build and distribute executables to all machines,
4. Automatically start execution and provide runtime monitoring facilities.

VPE is implemented on top of PVM [3], and many aspects of its use will be familiar to PVM programmers. Since PVM supports heterogeneous environments, so does VPE. However, VPE programmers do not directly use PVM primitives. Instead, VPE is designed to also be readily implemented on top of other message-passing libraries such as MPI [4].

1.1 Visual Representation

One of the hallmarks of VPE is that the programmer specifies the parallel structure of his or her program visually, by drawing a picture. VPE computations are graphs in which nodes represent processes, and arcs represent paths upon which messages flow from one process to another. Figure 1 shows an example VPE graph that multiplies two matrices (Cannon's algorithm). The programmer has entered a computation in C or Fortran for each node and these computations make explicit calls to VPE message-passing library routines to send and receive messages via the named "ports" that are attached to the nodes. This program will be fully explained in Section 3.

Visual representations have a number of advantages. They permit programmers to easily view and directly modify the structure of a program. Thus, programmers understand their programs' structure, and this is important since high performance depends upon careful structural design. Factors that programmers must keep in mind include what processes their program creates, where (machine and machine type) they

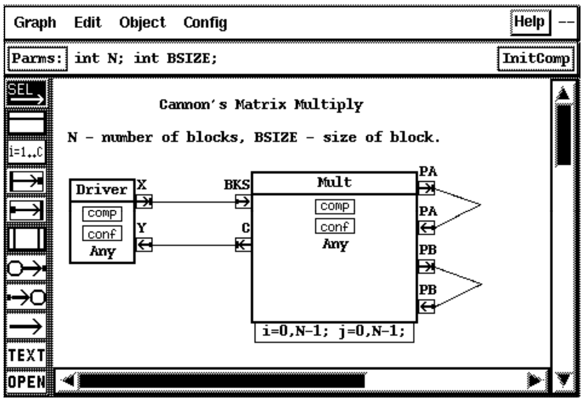

Figure 1: VPE Graph: Matrix Multiply

run, what computations the processes perform, which processes communicate with which other processes, the size of messages, the conditions under which messages are sent, and the granularity of the computation that takes place between interactions with other processes.

Graphs are a natural mechanism for organizing such information statically. Furthermore they lend themselves to the dynamic display via animation of runtime performance and structural data. Animation can be performed directly on the program as represented by the programmer. Programmers are not forced to manually relate animated displays to separate textual program representations.

VPE's visual program representation also lends itself to the development of a complete programming environment since computations are encapsulated in visual constructs. Such an environment can easily integrate and automate many of the steps in the programming process including program editing, program compiling, relating compile-time errors to specific attributes that are in error, program execution and debugging, and animation.

Many of these tasks are tedious (and serious stumbling blocks to novices) when using existing parallel environments and are even worse in heterogeneous distributed environments.

The VPE project has a number of goals. They are summarized below.

1. Allow users to specify process structure visually. Message sources and destinations are specified graphically.
2. Automate process creation at runtime. Programmers need not use explicit "spawn" calls.
3. Incorporate simple message-passing primitives to be called from node computations.
4. Be capable of using common message-passing libraries such as MPI and PVM as its execution target.
5. Support heterogeneous execution if the target message-passing library supports it.
6. Automate program compiles even in heterogeneous environments.
7. Automate program execution and relate runtime performance data and animation back to the user's original program representation.
8. Use a hierarchical name space to permit the creation of libraries (at the source level) in a simple manner that eliminates the need for complex context specifications.
9. Permit reuse of existing C and Fortran sequential subprograms.
10. Add little runtime overhead to what is already inherent in the target message-passing library.

1.2 Related Work: HeNCE and CODE

A number of other visual parallel programming languages and environments have been developed (see [6] for a survey). In fact, the authors are associated with the development of two previous systems, HeNCE [1], [2] and CODE 2.0 [5][6]. VPE and HeNCE and CODE have similar general goals, but the systems differ substantially in detail, just as HeNCE and CODE differ. The CODE model is dataflow oriented while HeNCE programs are (necessarily) structured "parallel flowcharts" in which nodes must declaratively specify access to shared variables in a global name space.

All three environments are based upon the idea that nodes represent computation, and arcs represent interactions (of some form) among nodes. HeNCE and CODE, however, are not based upon a traditional message-passing model. The fundamental difference between them and VPE is that HeNCE and CODE nodes represent sequential computations in which communications with other nodes occur only at the beginning and ending of the computation. Furthermore, these communications are expressed at a higher level of abstraction than are communications in VPE. HeNCE and CODE programmers make no explicit calls to message-passing library routines as VPE programmers must.

The HeNCE and CODE approach has many desirable properties. Nodes are essentially calls to sequential subroutines expressed in standard languages. The calls are embedded in an abstract visual specification of parallel structure. Programmers benefit from a separation of concerns. They first specify a set of sequential computations and then, separately, specify how they are to be composed into a parallel program. Also, the debugging process can be partitioned into the tasks of debugging a set of sequential routines and debugging the parallel interactions of the routines (which are then viewed as being atomic).

HeNCE and CODE arguably (it has not been demonstrated by implementation) enhance the portability of parallel programs in the sense of running well on multiple targets rather than the sense of running at all. This is because their models lends themselves to automatic analysis by intelligent translators. The program's sequential components are of known (or measurable) granularity and their interactions are specified at an abstract level that promotes analysis due to their direct and unambiguous representations. Such analysis is far more difficult for VPE programs since calls to communication routines are embedded within arbitrary C or Fortran computations.

The problem with the HeNCE/CODE approach is that it forces computations to be split into separate nodes when communications occur or when branching decisions control communications. This can result in complicated, awkward, and large graphs. Consider a simple imaginary computation in which F and G are sequential functions.

```
array A, B, C; scalar x = 0, y, z;
1: receive A from some process.
2: B = F(A);
3: Send parts of B to a set of processes, S;
4: For each process in S, receive z; x += z;
5: if (x < 0) receive y from one processes;
6: else receive y from some other process;
7: C = G(A, y, x);
```

Since communications cannot be embedded within HeNCE and CODE node computations, this program must be split into multiple nodes. In HeNCE a new computation node is required for lines 1, 4, 5, and 6 and four additional control flow nodes are needed as well. Thus eight nodes must be drawn, and six require annotation. Since communications are explicit (however abstract) in HeNCE and CODE, the programmer must state all communications such as the fact that the node running line 7 needs data from the processes running nodes 1, 4, and 5 or 6. This is wordy. CODE suffers from similar complexities involving multiple nodes or as few as one node that fires multiple times and has very complicated explicit firing conditions which are supplied by the programmer.

If F and G are truly large-grain routines that are logically decoupled, the HeNCE and CODE programs may be reasonable, but if they are not VPE's representation will be much simpler and more natural since all seven lines above may be regarded as pseudo-code for a single VPE node computation. Also, the computation G always follows F due to data dependences. The HeNCE and CODE implementations must perform analysis to determine that both should be run within a single processor to avoid the overhead of sending A. The VPE implementation will naturally do the right thing since VPE directly implements the process structure specified by the programmer.

Finally, we should note that the VPE model is a superset of the HeNCE and CODE models in the sense that it is possible for the user to choose to create nodes that communicate only at the beginning and end of computations. VPE is less abstract than HeNCE and CODE but provides greater expressive range. Its communications are specified less abstractly, but are simpler than those provided by most message-passing libraries since VPE uses graphical specification for the sources and sinks of messages. It is also closer to current programming practice. For better or worse, this suggests users will be comfortable with VPE since its learning curve is less steep.

2 Overview of VPE Environment and Language

Programs in VPE consist of a set of graphs which can call one another, thus permitting hierarchical program development much as subroutines do in conventional languages. Each graph contains computation (comp) nodes that represent processes that are specified as C and Fortran computations which contain calls to VPE message-passing routines. Messages flow on arcs that interconnect named ports attached to nodes. Message-passing calls reference port names. Also, comp nodes can be replicated, in which case instances are distinguished by integer-valued indices.

VPE's language is explicitly parallel. Programmers directly specify the parallel structure of their programs and must choose appropriate parallel structures in order to achieve good performance. VPE's visual representations assist in this task.

The VPE environment itself runs on UNIX workstations under X windows, although the parallel programs created within it may be executed on different types of machines. VPE programs consist of several elements each of which is stored in a separate file.

1. There is one project file (ending in ".proj") that lists all of the graphs in the program. There is thus one project file per program.

2. There is a graph file (ending in ".gr") for each graph in the project.

It is possible for a single graph file to be included in multiple projects. Thus, graph files may be stored in libraries. It is also possible for a graph file to exist without being in any project file. Such a graph is simply not used in any program at the moment.

2.1 The VPE Graphical User Interface

When VPE is run, one or more windows will appear on the workstation screen. The project window will always be open and will display the contents of the project file and the state of the virtual parallel machine. In addition, zero or more graph edit windows will display graphs. VPE's graphical user interface is implemented using the Tcl/Tk toolkit [7] developed by John Ousterhout.

2.1.1 The Project Window

The project window (Figure 2) serves three purposes. First, it lists the names of the graphs that are a part of the current project. It is these graphs (not the set of graphs that are open in edit windows) that will be translated to form a complete parallel program. Second, it lists all hosts in the current virtual parallel machine as well as providing a place to show host state during program execution. Finally, it contains an output area for output from parallel programs and messages from VPE.

Menu picks in the project window allow both the list of graphs in the project and the virtual parallel machine to be managed. For example, users can add and delete items to the respective lists. VPE has keyboard alternatives for all frequently selected menu items.

2.1.2 Graph Edit Windows

Graphs are viewed and edited in graph edit windows such as the one shown in Figure 1. VPE users can open graphs even if they are not in the project since they may wish to view a graph in another project while

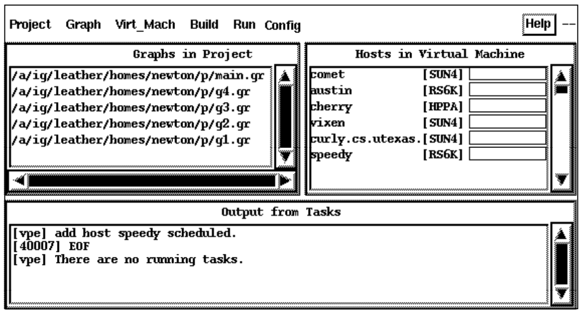

Figure 2: VPE Project Window

working on the current one. Or, they may wish to cut and paste nodes between graphs from different programs.

The VPE graph editor is designed to be familiar to users of popular personal computer based drawing programs. Users select tools with the mouse from the toolbar on the left and then use the tool in the white space drawing area on the right. For example, to draw a comp node, the user selects the Comp tool (second down) and then clicks on white space. The Select tool (top) is used to select objects to cut, copy, delete, move, resize, etc. It is possible to select multiple objects simultaneously. The Open tool is used to open attribute forms for an object.

2.1.3 Attribute Windows

Many objects in VPE have attributes that the user must be able to view and edit. Other than required node names, attributes are edited in attribute windows which are opened by clicking the Open tool on an object or special part of an object. For example, to enter the computation of a computation node, the user clicks the Open tool on the box labeled "comp" shown on all comp nodes.

2.2 The VPE Language

Programs in VPE consist of nodes of various kinds, arcs, and textual annotations. There are four types of nodes and Figure 3 shows them all. Three of the four are used in hierarchical program structuring, allowing one graph to call another.

2.2.1 Compute Nodes

Compute (comp) nodes represent processes and are displayed as variable sized boxes with single vertical

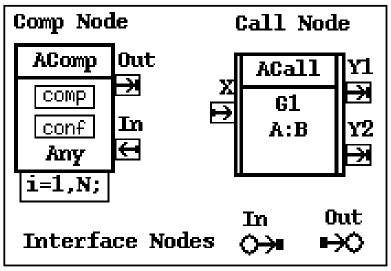

Figure 3: VPE Node Types

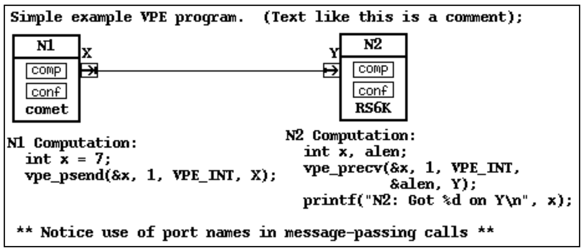

Figure 4: Simple Example

lines on the left and right side. Messages enter and leave comp nodes via named input and output ports. The programmer may add any number of ports to a comp node. The programmer may also add a replication box to the node. This permits multiple copies of the node to be run. The copies are distinguished by integer valued indices.

Comp nodes have several attributes which the programmer must set. The node's computation is the most important. It consists of an ordinary C or Fortran subprogram body which contains calls to the VPE message-passing library. These calls send messages out on ports or receive them from them.

Figure 4 shows a trivial example program as well as the text of the nodes' computations. Node N1 sends a single message containing an integer to node N2. The message leaves port X and enters port Y since an arc leads from one to the other. Notice that the node computations mention only local port names. Thus, nodes are "black boxes" that can be wired (by arcs) into any graph they are pasted into. The key calls are to vpe_psend and vpe_precv.

```
vpe_psend(Data, NumToSend, Type,
          PortToSendOn);
vpe_precv(Data, DataSize, Type, NumRecvd,
          PortToRecvOn);
```

The VPE library contains many other routines as well. They are generally reminiscent of PVM and include buffer management, message packing and unpacking, and multicast routines.

The programmer may also specify the computer (or computer type) that a node is to run on. In Figure 4, N1 runs on a machine called "comet" and N2 runs on any IBM RS/6000 that is in the current virtual parallel machine. A value of "any" allows a node to run on any host in the virtual parallel machine. When it must choose, VPE allocates nodes to machines in a simple round robin fashion.

Finally, the programmer must supply replication expressions for all replication boxes that have been added to comp nodes. The syntax is

```
IndVar = <FromExpr>, <ToExpr>; or
IndVar = <FromExpr>, <ToExpr> by <StepExpr>;
```

For example, one could enter the following expression to create a two-dimensional array of nodes, all running in parallel.

```
i = 1, N+1; j = 1, M;
```

Variables i and j are may be referenced in the node's computation. They will contain the values of the node instance's indices. N and M may be constants or graph parameters- variables that can be read from anywhere in the graph, but can be given values only at graph creation time. Parameters are discussed in detail in Section 2.2.2.

2.2.2 Call and Interface Nodes

Call nodes permit one graph to call another. In this way, VPE supports hierarchical program development. Call nodes have ports which correspond to actual parameters. Graphs that are called have input and output interface nodes which are formal parameters. There is always a port on the call node for every interface node in the graph it calls. No recursion is allowed in calls, and calls have inlining semantics.

Figure 5 shows an example of one graph (foo) calling another (bar) by means of a call node named Call1. A message that leaves port $Y1$ in foo goes to port IN of the call node and then to input interface node IN in graph bar. From there, it follows the arc to port X on comp node Bar_1. Similarly, messages that leave this node's port Y, end up at port Z on comp node Cat.

Graphs also have attributes called "parameters" and "initialization computations." Parameters are variables that are given values by the programmer's

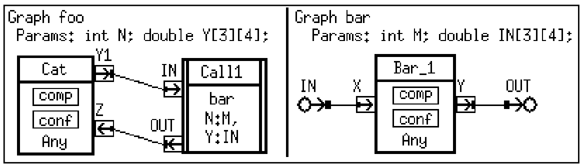

Figure 5: Example Call

initialization computation before any comp node processes are started. Then, the parameters can be accessed as ordinary variables from within comp node computations. In other words, first the initialization computation runs giving values to parameters, then these values are broadcast to all comp node before they begin to run. This is a convenient way to broadcast graph-wide information such as array sizes. Parameters are so named because they parameterize graphs. Once a comp node is running, changes it makes to a parameter are not propagated to other nodes; each node has a local copy.

Parameters can also optionally be bound in calls. Figure 5 shows foo's parameters N and Y bound to bar's parameters M and IN. Since foo calls bar, its initialization computation is run first. Then, as bar's initialization computation starts, M and IN have the same values as N and Y.

3 Example program

In this section, we return to the matrix multiplication program shown in Figure 1 and explain it in full. It demonstrates a computation involving a replicated node.

The program is based on Cannon's algorithm, a space-efficient algorithm in which the input matrices A and B are divided in block fashion on an logical N x N grid of processors. Each logical processor will allocate storage for only a single block of A, B, and the output matrix C. The algorithm will involve communication steps in which entire blocks are sent from one processor to another. The algorithm is as follows.

1. Every processor (i, j) shifts its block of A by i processes to the left and its block of B by j processes up, wrapping in both cases.

2. Repeat for N times, each processor (i, j) forms $C = C + A * B$, where A, B, and C, are the blocks it currently holds, and then it shifts its block of A one processor up and its block of B one processor left.

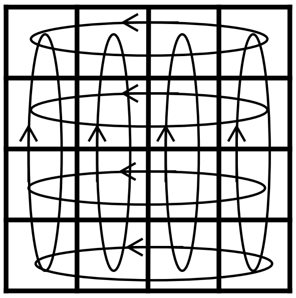

Figure 6: Cannon's Algorithm

Upon completion, each processor contains its block of C. Figure 6 shows an N x N array of processors and the step two shifting patterns of A and B.

This algorithm is quite easy to express in VPE. The program contains two nodes. Node Driver is not replicated, and node Mult is replicated N^2 times in a two dimensional grid fashion, as its replication shows in Figure 1. Driver simply assigns values to A and B and then distributes blocks of them to the appropriate instances of Mult. Finally, it awaits a message from each Mult instance containing a block of C.

Driver's sending phase is shown below. Blocks do not occupy either contiguous storage or storage at a fixed stride in conventional arrays so each row of a block is packed separately into the message by calls to vpe_pkdouble. $BSIZE$ is the length of a row in a block. Notice that it and N are parameters in the graph. Their values are set in the graph's initialization computation.

The message containing the blocks of A and B is sent by the call to vpe_send. Here, X is the name of the port the message will exit, and i and j are the indices of the receiving instance of Mult. Notice that there is an arc from port X to port BKS. The instances of Mult will receive their blocks on the latter port.

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    /* Pack a block of A and B */
    vpe_initsend(VpeDataDefault);
    for (k = 0; k < BSIZE; k++)
      vpe_pkdouble(&A[(i*BSIZE+k)*r+j*BSIZE],
                   BSIZE, 1);
    for (k = 0; k < BSIZE; k++)
      vpe_pkdouble(&B[(i*BSIZE+k)*r+j*BSIZE],
```

```
                BSIZE, 1);
    /* Send blocks to node Mult[i][j] */
    vpe_send(X, i, j);
  }
```

Driver's code to receive the blocks of C follows. It is roughly the reverse of its send code, but only one block is in each message. Arguments i and j in the vpe_recv call are the indices of the sending instance of Mult. The message arrives on port Y.

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    /* recv block of C from Mult[i][j] */
    vpe_recv(Y, i, j);
    /* Unpack block of C */
    for (k = 0; k < BSIZE; k++) {
      ind = (i*BSIZE+k)*r+j*BSIZE;
      vpe_upkdouble(&C[ind],
                    BSIZE, 1);
    }
  }
```

There are three phases to the Mult nodes' computation. The index variables in Mult's replication are i and j, thus these variables are constants in the node that identify its index. In the first phase, Mult receives a message containing its block of A and B and then performs the initial shift (which Driver could have done). Notice that there are no indices in the vpe_recv(BKS) call. This is because Driver is not replicated.

VPE psend and precv calls are used for the communication of blocks among the instances of Mult. These combine all necessary initialization, packing, and communication operations into one call. In all cases, the last two arguments are the indices of the other node involved in the communication.

```
vpe_recv(BKS);  /* recv blocks of A, B */
vpe_upkdouble(A, n, 1);
vpe_upkdouble(B, n, 1);

/* initially shift A[i][j] by i processes to
   the left and B[i][j] by j processes up,
   wrapping in both cases */

vpe_psend(A, n, VPE_DOUBLE, PA,
          i, pmod(j-i, N));
vpe_precv(A, n, VPE_DOUBLE, &alen, PA,
          i, pmod(j+i, N));
vpe_psend(B, n, VPE_DOUBLE, PB,
          pmod(i-j, N), j);
vpe_precv(B, n, VPE_DOUBLE, &alen, PB,
          pmod(i+j, N), j);
```

Step two of the algorithm involves multiplication of the local blocks followed by block shifting among the Mult instances, all within a loop. Notice that there are input and output ports called PA and PB for the block shift messages.

```
for (k = 0; k < N; k++) {
  MultBlocks(A, B, C, BSIZE);
  if (k == N-1) break;  /* we are done */

  /* shift A one left and B one up */
  vpe_psend(A, n, VPE_DOUBLE, PA,
            i, pmod(j-1, N));
  vpe_precv(A, n, VPE_DOUBLE, &alen, PA,
            i, pmod(j+1, N));
  vpe_psend(B, n, VPE_DOUBLE, PB,
            pmod(i-1, N), j);
  vpe_precv(B, n, VPE_DOUBLE, &alen, PB,
            pmod(i+1, N), j);
}
```

Finally, each Mult instance sends its block of C to Driver. There are no indices in the vpe_psend call since Driver is not replicated.

```
/* send block of C to Driver */
vpe_psend(C, n, VPE_DOUBLE, C);
```

Since one of the virtues of Cannon's algorithm is to allow each node to contain storage for only a single block of A, B, and C, it is wasteful to have node Driver gather them all in one place (unless needed for some other reason). This could be avoided by having the Mult nodes do their own I/O, and this could be done using UNIX facilities assuming execution on a workstation network. However, this example points out the value an integrated parallel I/O library would have for VPE and many other message-passing systems.

4 Building and Running Programs

VPE programs are built by picking "Build" from the project window's Build menu. VPE will first translate all graphs in the project into C or Fortran (with PVM calls) and then distribute this source to all necessary machines. It will then compile the source on all of the needed machine types (in parallel). Error messages are displayed in project window's output area.

The program can then be run by picking "Run" from the project window's Run menu. PVM tasks will be started on all necessary machines, and output written to stdout and stderr will be gathered and displayed in the output area. VPE also allows programs to be run manually, when VPE itself is not running. There is no animation or host status display in this case.

5 Animation

A runtime program animation facility is planned for VPE. This animation will be performed directly on the programmer's VPE graphs. It will include highlighting arcs when messages are sent on them as well as host utilization displays in the boxes to the right of host names in the project window.

6 VPE Versus PVM

The goal of VPE is to reduce the complexity of parallel programs without paying large overheads. VPE's success can be measured by comparing it to lower-level programming, such as simply writing programs with PVM. This section presents some comparisons for the matrix multiply example (N = 2, BSIZE = 100).

In terms of some simple complexity measurements of the program text the user must write, the VPE program is substantially simpler than its PVM equivalent. The PVM program requires 67% more lines (comments removed for all) and 85% more library (vpe_ and pvm_) calls than the VPE program. Of course, this is a small example.

VPE's performance on this program is not measurably slower since it maps quite directly to PVM. VPE spawns one more task (for initialization computations) and sends N+1 more messages (for initialization).

7 VPE Implementation Status

At the time of this writing, most aspects of VPE are complete, but implementation is ongoing. The following items are implemented: program editing, annotation, and drawing, virtual machine management and dynamic host list display, program building (assuming all machines share a common file systems), program execution with output capture, and online help. The examples in this paper and many others build and run with no manual intervention. Items remaining include program animation and status display of the hosts in the virtual machine. Also, building in heterogeneous environments needs work (especially for cases where a shared file system is not available). Implementation of the latter is facilitated by experiences with HeNCE and by the use of PVM itself as file transport and remote execution (of build commands) facility. The portability and support for heterogeneous computation of PVM itself is also a key enabling technology.

Contact newton@cs.utk.edu for information on how to obtain VPE. It is free and available to all in source or binary form.

Acknowledgments

This research is supported in part by NSF grant NSF-ASC-9214149 and by PICS subcontract 11B99737C.

References

[1] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam, "Graphical development tools for network-based concurrent supercomputing," *Proceedings of Supercomputing 91*, pp. 435-444, Albuquerque, 1991.

[2] A. Beguelin, J. Dongarra, G. A. Geist, and V. S. Sunderam, "Visualization and Debugging in a Heterogeneous Environment," *IEEE Computer*, Vol. 26, No. 6, June, 1993.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994.

[4] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *Journal of Supercomputing Applications*, Vol. 8, No. 3/4, 1994.

[5] P. Newton and J.C. Browne, "The CODE 2.0 Graphical Parallel Programming Language," *Proc. ACM Int. Conf. on Supercomputing*, July, 1992.

[6] P. Newton, "A Graphical Retargetable Parallel Programming Environment and Its Efficient Implementation," Technical Report TR93-28, Dept. of Computer Sciences, Univ. of Texas at Austin, 1993.

[7] J. Ousterhaut, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA,1994.