Overview of VPE: A Visual Environment for Message-Passing Parallel Programming†

Peter Newton
Jack Dongarra

Abstract

VPE is a visual parallel programming environment for message-passing parallel computing and 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. VPE is designed to be implemented on top of standard message-passing libraries such as PVM and MPI.

1.0 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. VPE is a visual programming environment that is intended to provide a simplified interface to the process of creating message-passing parallel programs. The environment will assist the programmer in creating, compiling and running parallel programs (even in heterogeneous environments) and will then animate the program's execution. VPE is designed to be readily implemented on top of common message-passing libraries such as PVM [Gei94] and MPI [MPI94]. Such libraries target both MPP systems and networks of workstations.

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 message flow from one process to another. Figure 1 shows a simple example VPE graph that computes the dot product of two vectors and is heavily commented. 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 3.0.

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, 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.

† This research is supported in part by NSF grant NSF-ASC-9214149 and by PICS subcontract 11B99737C S-77, mod. 2.

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.

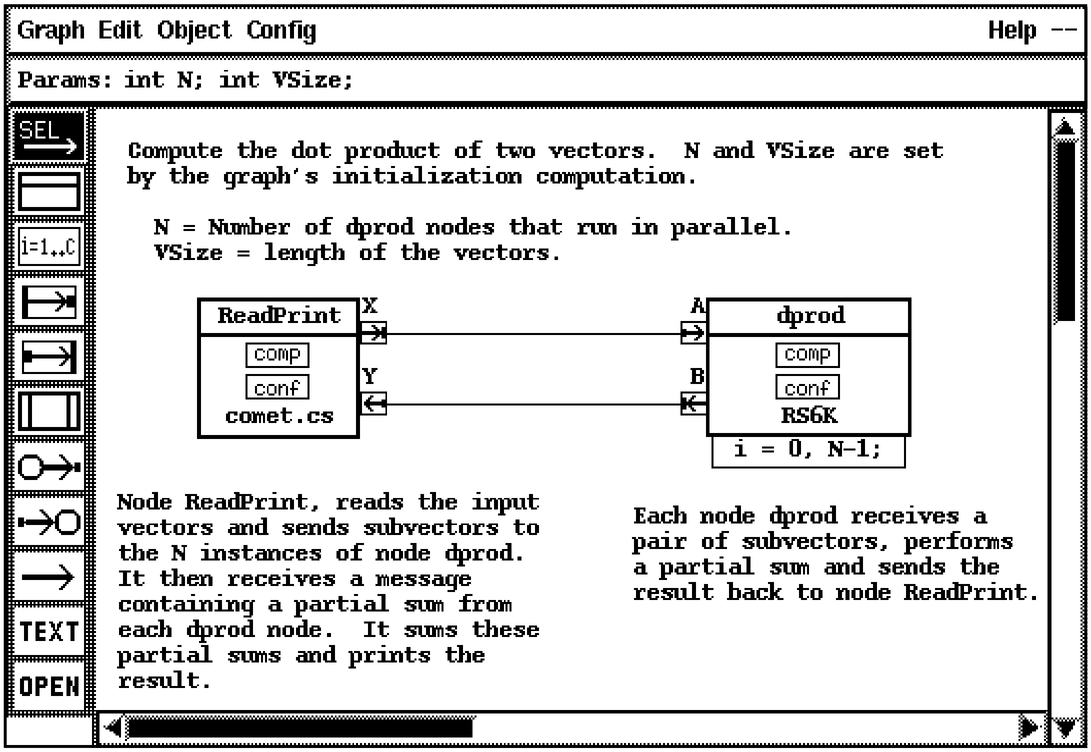

Figure 1. Example VPE Graph (Dot Product Computation).

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 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. VPE allows users to (for example) automatically compile all parts of a program on all necessary machine types via a single menu pick. It is a complete integrated development environment.

The VPE project has many goals. They are summarized below.

1. Allow users to specify process structure visually. Message sources and destinations are specified graphically.
2. Relate runtime performance information and animation directly to this user-specified structure.
3. Automate process creation at runtime. Programmers need not use explicit "spawn" calls.
4. Incorporate simple message-passing primitives to be called from node computations.

5. Be capable of using common message-passing libraries such as MPI and PVM as its execution target.
6. Support heterogeneous execution if the target message-passing library supports it.
7. Automate program compiles even in heterogeneous environments.
8. Automate program execution and relate runtime performance data and animation back to the user's original program representation.
9. 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.
10. Permit reuse of existing C and Fortran sequential subprograms.
11. 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 programming languages and environments have been developed (see [New93] for a survey). In fact, the authors are associated with the development of two previous systems, HeNCE [Beg91, Beg93] and CODE 2.0 [New92, New93]. 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 = 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 pseudocode 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 that 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.

2.0 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 (GUI).

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 zero or more graph edit windows will display graphs. In addition, users can open attribute windows to view and change attribute values of nodes, etc. These windows are generally non-modal so that information from multiple sources can be simultaneously viewed or manipulated, and objects can be cut and pasted between graphs.

VPE's graphical user interface is implemented using the Tcl/Tk toolkit [Ous94] developed by John Ousterhout.

2.1.1 The Project Window

As stated above, the project window simply 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. Figure 2 shows the VPE project window.

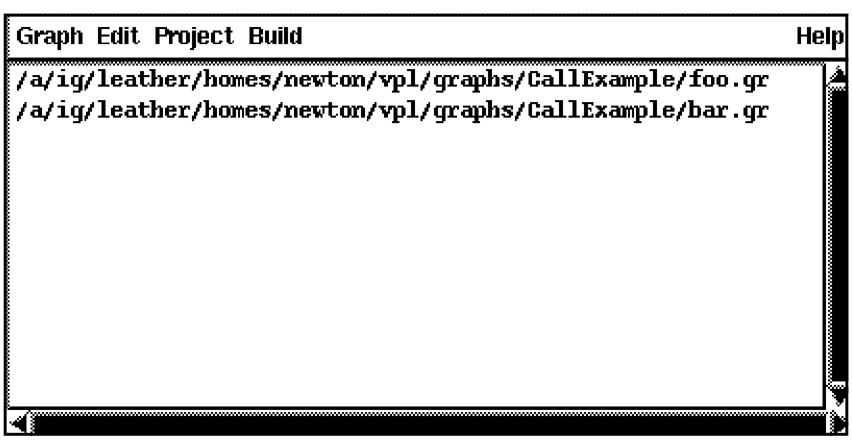

Figure 2. A VPE Project Window Showing Two Graphs in the Project,

Menu picks from the project window permit graphs to be added to and removed from project, build an executable program from the project, run the project, etc. In addition, users can easily open graphs in the project in a graph edit window by selecting one or more graphs with the mouse and picking "Open Selected" from the Graph menu or pressing Control-L. VPE has such 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 3. 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 working on the current one. Or, they may wish to cut and paste nodes between graphs from different programs.

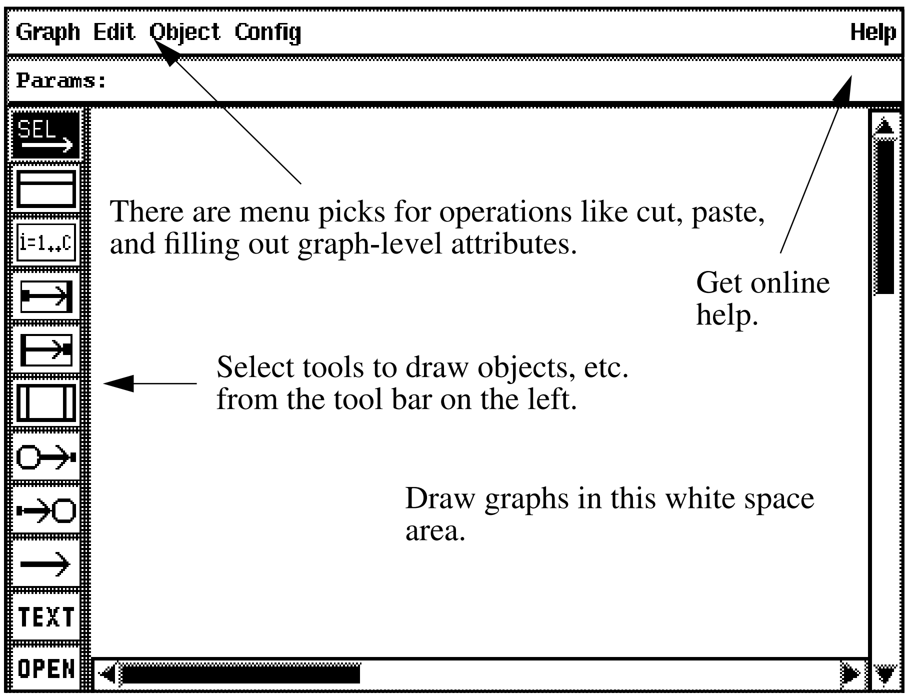

Figure 3. An Empty Graph Edit Window

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. Figure 4 summarizes VPE's graph edit tools.

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 non-model attribute windows.

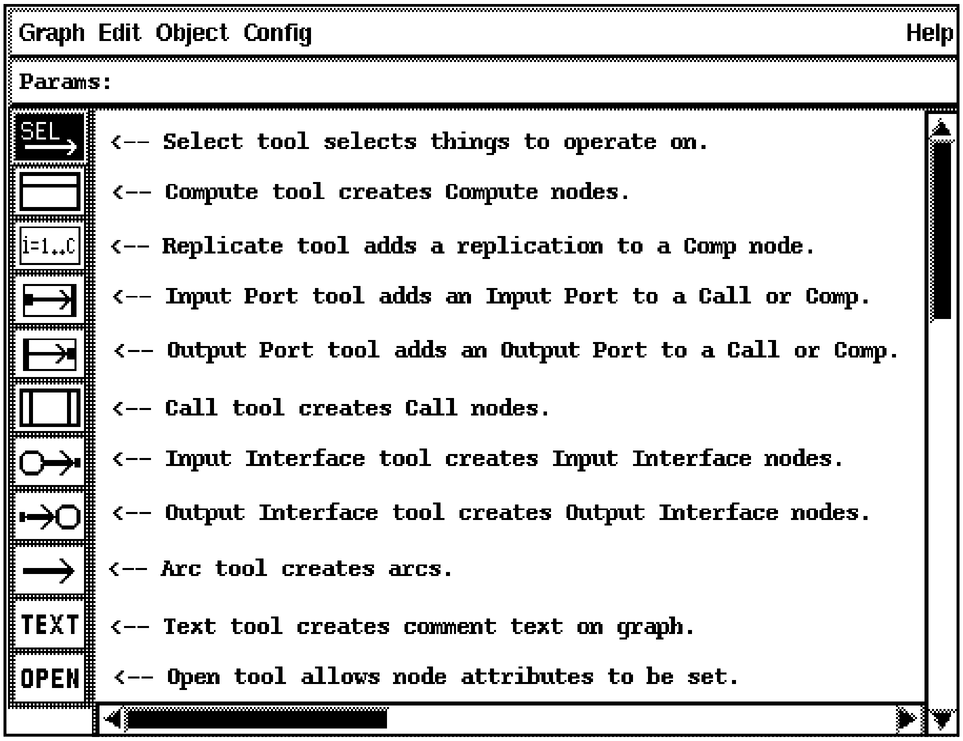

Figure 4. VPE Graph Edit Tools.

Attribute windows 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 labelled “comp” shown on all comp nodes. Figure 5 shows a typical attribute window. This paper will define the attributes of VPE objects, but will not focus on the details of attribute forms.

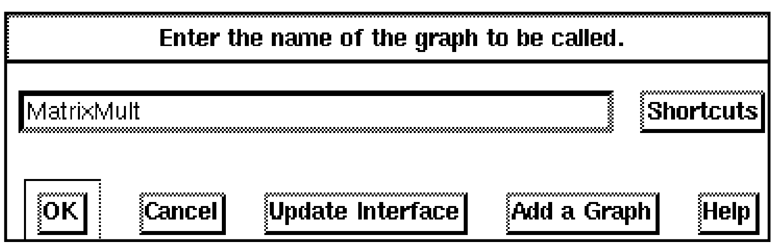

Figure 5. A Typical VPE Attribute Window.

2.2 The VPE Language

Programs in VPE consist of both textual annotations and visual constructs including arcs and various types of nodes. There are four types of nodes in VPE and Figure 6 shows them all.

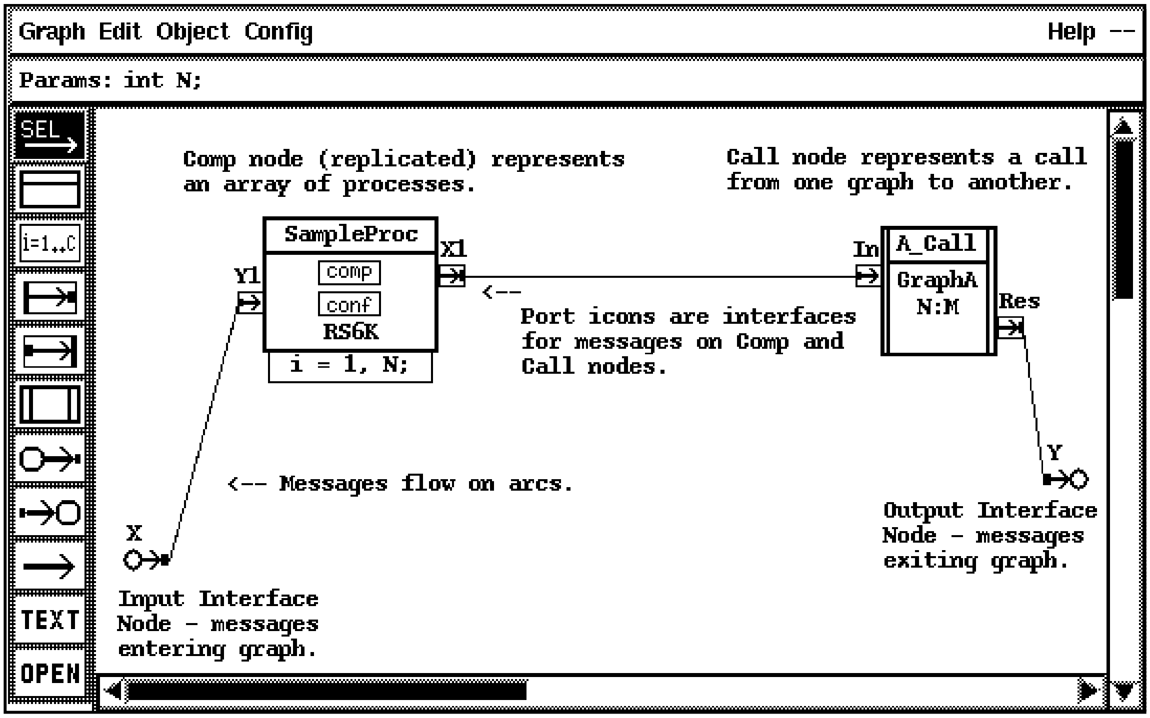

Figure 6. Sample Graph Showing VPE Node Types.

2.2.1 Compute Nodes

Compute (comp) nodes represent processes and are displayed as variable sized boxes with single vertical lines on the left and right side. Comp nodes contain several fields that represent attributes. They are listed here from top to bottom and are set by clicking the Open tool on them.

1. The name of a comp node is a comment, but it is a good idea to give comp nodes meaningful names since error messages refer to them.
2. The Comp button holds the C or Fortran text of the node's computation.
3. The Conf button holds various node configuration options including a list of C or Fortran files that must be compiled with the node since it calls routines in them.
4. The Architecture field shows the name of the machine or machine type the node is to execute on.
5. Replication expressions (optional).

<u>Comp Node Replication</u>

A comp nodes can be replicated by adding a replicator box to it by clicking near the node with the replicate tool. A replication will appear. Click on it with an open tool and enter one or more replication expressions. For example, one could enter

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

to create a two-dimensional array of nodes, all running in parallel. 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 3.0 and Section 4.0.

Input and Output Ports

Message that arrive at a comp node enter via named input ports and messages that leave exit via named output ports. Ports are added to comp nodes using the two port tools. Each port must be given a name and no node may have two input ports with the same name or two output ports with the same name.

Calls to message-passing routines inside node computations refer to port names. VPE's message-passing routines will be listed in Section 5.0, but Figure 7 shows a simple example in which a single value of type double is sent from one comp node to a second comp node. The nodes' computations are in C and are shown below the nodes.

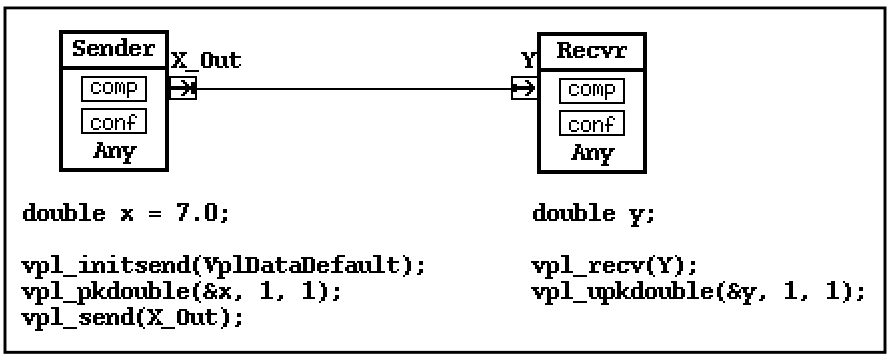

Figure 7. Example Send.

Node Sender first initializes a data send buffer and then packs a double into it. It then sends a message containing the buffer out on port X_Out. Port X_Out is connect by an arc to port Y of node Recvr. This node receives a message from its port Y and unpacks a double from it.

Section 3.0 will present an example in which the receiving node is replicated (with a single replication expression). In that case, the sending node would execute

```
vpe_send(X_Out, i);
```

where *i* contains the index of the receiving node. This type of send would usually be placed inside a loop in which *i* takes on a sequence of values.

2.2.2 Call and Interface Nodes

Call nodes represent calls from one graph to another and are represented as variable sized boxes with double vertical lines on the left and right side. Figure 8 shows an example of a call node and the graph it calls (G2). G2's interface nodes define its interface. There is an exact correspondence between the interface nodes in a graph and the ports on a call node that calls it. When a message is sent on an arc to the call node's port A, that message "appears" on input interface node A and goes to port X1 due to the arc from A to X1. The other ports act similarly.

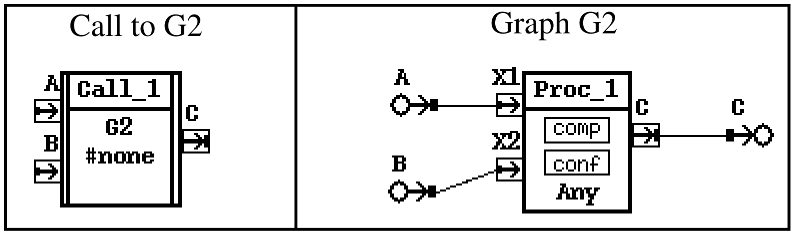

Figure 8. Call Node and Graph Called from It.

It is also possible to bind parameters from the calling graph to parameters of the graph that is called. This will be described in Section 4.0.

3.0 Example program

In this section, we return to the dot product program shown in Figure 1 and explain it in full. The program works as follows.

1. There is a single instance of comp node ReadPrint and there are N instances of comp node dprod which are distinguished by differing index values (dprod[0] ... dprod[N-1]). All comp nodes instances are processes that run concurrently.
2. ReadPrint first reads values for two vectors and then sends subvectors from each vector to each of the dprod nodes.
3. The dprod nodes wait to receive their subvectors from ReadPrint. When they receive them, they compute the dot product of their subvectors, forming a part of the final dot product. Each dprod node sends this value back to ReadPrint.
4. ReadPrint waits to receive messages from the dprod nodes with the dot products of the subvectors. A running sum is kept as each message arrives. When N messages have arrived, the dot product is complete.

3.1 Parameters and Graph Initialization Computations

As mentioned above, graph parameters are variables that are assigned a value by a graph's initialization computation and can then subsequently be read from any node in the graph. The "Params:" line at the top of Figure 1 shows that the programmer has declared two integer parameters for the graph: *N* (the number of dprod instances to create) and *VSize* (the length of the input vectors to read). Both are initialized by the graph's programmer-supplied initialization computation which runs before any of the comp nodes in the graph are instantiated. When the initialization computation is complete, comp nodes are instantiated, and they may then read the values of *N* and *VSize*. The name "parameter" is intended to suggest parameterization of a graph at the time it is created.

It is legal for nodes to write to parameters but changes will *not* be propagated to other node instances. Parameters are, in fact, node-local variables that happen to be initialized by the graph's initialization computation.

Parameters play a vital role in replicating comp nodes. They are the only variables other than the index variables themselves that may appear in replication expressions. Thus, in VPE one must decide at the beginning of a computation how many node instances will be created since parameter values used in replication expressions must be set in graph initialization computations. This is not a serious limitation for many algorithms since the number of instances to create is often dictated by the number of processors in the parallel machine on which the program will run.

Section 4.0 will describe how a call node can initialize parameters in the graph it calls using values of parameters in the graph containing the call node. A graph's parameters together with its interface nodes completely define its interface.

3.2 Comp Node Specifications

<u>Comp Node ReadPrint</u>

Figure 1 shows that ReadPrint has been configured to run on a particular machine named "comet.cs" that is on a network of machines being used as a virtual parallel computer. It has an output port X and an input port Y. ReadPrint's complete computation follows. Pay particular attention to the calls to vpe_send and vpe_recv. The other "vpe_" calls manage message buffers.

```
double *V1 = MkVect(VSize), *V2 = MkVect(VSize), Sum = 0.0, PartSum;
int i, Start = 0, Len = VSize/N+1, Extras = VSize % N;

ReadVect(V1, VSize);                 /* Read vectors from somewhere.     */
ReadVect(V2, VSize);

for (i = 0; i < N; i++) {            /* for each i from 0 to N - 1, pack  */
   if (i == Extras) Len -= 1;        /* and send subvectors on port X     */
   vpe_initsend(VpeDataDefault);     /* with node index i.  Since X leads */
   vpe_pkdouble(&V1[Start], Len, 1); /* to port A on dprod, dprod[i] will */
   vpe_pkdouble(&V2[Start], Len, 1); /* receive the message.              */
   vpe_send(X, i);
   Start += Len;
}

for (i = 0; i < N; i++) {            /* Wait for a message with partial   */
   vpe_recv(Y, VpeAny);              /* sum to arrive from each of        */
   vpe_unpkdouble(&PartSum, 1, 1);   /* the N dprod nodes.  Messages can  */
   Sum += PartSum;                   /* be received in any order.         */
}

/* Now Sum contains the dot product */
```

The first loop sends subvectors of *V1* and *V2* to all dprod nodes. The code involving variable Len makes sure things work when *N* does not evenly divide *VSize*. The message containing the subvectors is sent to node dprod[i] by the vpe_send(X, *i*) call. Here, X refers to the output port the message will be sent on and *i* refers to the index of the receiving node. Since there is an arc from port X to port A, this message will be received on port A of dprod[*i*].

The second loop receives a dot product of subvectors from each dprod instance. The key call is to vpe_recv(Y, VpeAny). The use of value VpeAny for the index of the sending node allows the messages to be received from the *N* dprod instances in any order. If, instead, vpe_recv(Y, *i*) were called then ReadPrint would receive messages in order first from dprod[0], then dprod[1], and so on.

<u>Comp Node dprod</u>

Figure 1 shows that the dprod is replicated *N* times with index variable *i* taking on values from 0 to *N*-1. Inside dprod's computation, *i* will hold the index of the node instance. Furthermore, dprod has been configured so that all instances will run on available machines of type RS6K (IBM RS/6000 workstations). Node dprod's computation follows.

```
double *SV1 = MkVect(VSize/N+1), *SV2 = MkVect(VSize/N+1),
       PartialSum = 0.0;
int j, Len = VSize/N+1;

if (i >= VSize % N) Len -= 1;        /* Get the size right              */

vpe_recv(A);                         /* Receive and unpack subvectors   */
vpe_unpkdouble(SV1, Len, 1);         /* from ReadPrint.                 */
vpe_unpkdouble(SV2, Len, 1);

for (j = 0; j < Len; j++)            /* Form the partial sum.           */
   PartialSum += SV1[j] * SV2[j];

vpe_initsend(VpeDataDefault);        /* Send partial sum out on port    */
vpe_pkdouble(&PartialSum, 1, 1);     /* B, which is connected to port Y */
vpe_send(B);                         /* on ReadPrint.                   */
```

The dprod instances first receive a message containing subvectors of V1 and V2 by means of the call vpe_recv(A). There are no node indices since ReadPrint is not replicated. It then computes the dot product of the subvectors and sends it back to ReadPrint via vpe_send(B). Again, there are no node indices because ReadPrint is not replicated.

3.3 Summary: Steps in Creating the Dot Product Program

This section summarizes the steps that a programmer must perform in order to create the dot product program. They do not necessarily have to be done in this order.

1. Draw all nodes, ports, replication boxes, and arcs in the graph. Name the ports.
2. Declare the parameters *N* and *VSize*.
3. Enter the graph's initialization computation. It must give values to the parameters.
4. Name the comp nodes (optional).
5. Choose the machine or machine type the comp nodes will run on. (Details vary according to the target machine. The default is to permit the node instances to run on any available processor).
6. Enter dprod's replication expression.
7. Enter the computations for ReadPrint and dprod.

At this point, the program is complete. It can be compiled, and executed.

4.0 Graph Calling

VPE supports hierarchical program development and the creation of libraries of graphs. One VPE graph can call another by means of a call node, but neither direct nor indirect recursion is allowed. Calls have "inlining" semantics-- programmers can think of call nodes as being replaced by the graph they call. It is possible for the same graph to be called from two or more different call nodes, but a separate instance of the graph will be called from each.

4.1 Graph Interfaces

There are two aspects to a graph's interface, interface nodes (input and output), and parameters. They serve very different purposes.

Interface nodes are points through which messages enter and leave graphs. There will be a port on a call node that calls a given graph for every interface node in the graph. There is a one-to-one correspondence between ports on call nodes and interface nodes in the graphs they call. When a message enters (or leaves) a port on a call to goes to (or comes from) the corresponding interface node. Naturally, input interface nodes pair with input ports and output interface nodes pair with output ports.

All interface nodes must be named by a legal VPE identifier and no two input interface nodes in the same graph may have the same name. Similarly, no two output interface nodes in the same graph may have the same name. VPE also requires that all ports have an arc incident upon them. This implies that all interface nodes in a graph will have an actual parameter bound to them by means of an arc incident upon the corresponding port in a call node.

Notice that ports and interface nodes are not typed. This is because the messages that flow through them are not typed, and may indeed have values of several types packed within them. Thus, type errors are manifested within computations where the programmer might unpack the wrong type from a message.

Graph parameters are variables that are set in graph initialization computations and can then be read anywhere in the graph, but they also play a role in graph calling. Programmers can *optionally* bind a parameter in a calling graph to a parameter in the graph it calls. This binding is an attribute of a call node. Figure 9 shows graph main's parameter *M* bound to graph addN's parameter *N* by means of the expression "*M*:*N*" in call node C1. When parameters are bound, types must match exactly (array sizes must match).

Bound parameters work in the following way. Let graph G1 have a parameter *x* and let G1 call graph G2 by means of a call node in G1. Furthermore, let G2 have parameters *p* and *q*. Assume that G1's call node binds *x* to *p* by means of "*x*:*p*" but binds nothing to *q*.

Execution will begin with G1's initialization computation running to completion. It is expected to give a value to x. Then, G2's initialization computation can begin, but in it *p* is given the value of *x* initially because of the binding. G2's initialization computation is then free to optionally modify *p* but is expected to give a value to *q* since it not bound to a parameter in G1. When G's initialization computation is complete, all comp nodes can be instantiated. Programmers, however, should not assume that all initialization computations are complete before any comp nodes in the program are instantiated, but they may assume that all parameters will given values from initialization computations before they can be used by a comp node, even in a replication expression.

4.2 Call Example

This section presents a very simple example of one graph (main) calling another (addN) in which both interface nodes and parameters are bound. Both graphs are shown in Figure 9. Graph addN's purpose is to receive a stream of messages from its input interface node X. Each message contains an integer. It then adds the value of its parameter *N* to the integer and sends it back to main via its output interface node Z.

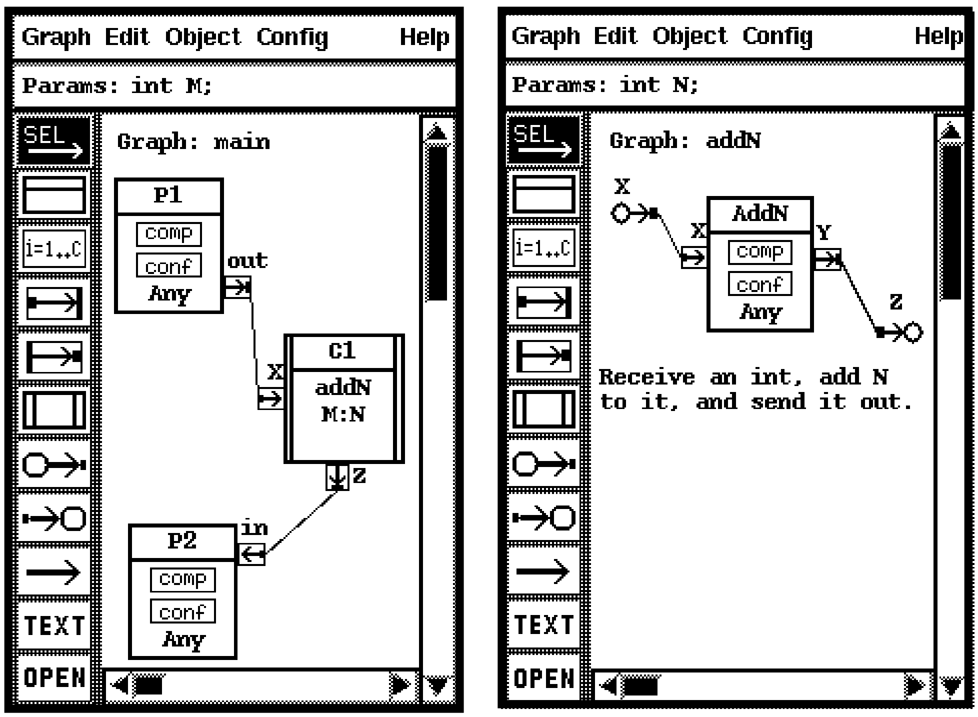

Figure 9. Simple Graph Call Example.

Graph addN's parameter *N* is bound to main's parameter *M*. Thus, as long as addN's initialization computation does not change the value of *N*, it will have the value given to *M* in main's initialization computation.

On the other hand, it is legal for the programmer to choose to alter *N* in addN's initialization computation. It could completely override the value of *M* via

```
N = 10; /* Give N a fixed value, regardless on M */
```

or it could change *N* in a way that still makes use of *M*'s value.

```
N = N + 1; /* N ends up with value M+1 */
```

In the former case, of course, one wonders why the programmer bothered to bind *M* to *N* in C1 since it has no effect, but the program is legal as written.

5.0 Message-Passing Primitives

This section describes the communications primitives that can be called from within VPE computations. Collective operations apply to all replications of a single node instance. Additonal routines may be added to VPE as needed.

Buffer Management Routines

```
vpe_initsend(encoding);
     Clear send buffer.

vpe_pkTYPE(address, NumItems, Stride);
     Pack values into message buffer.

vpe_upkTYPE(adress, NumItems, Stride);
     Unpack values from buffer.
```

Communication Routines

```
vpe_send(PORT_NAME, index, ...);
     Send current buffer.

vpe_psend(PORT_NAME, index, ..., data, length, type);
     Pack and send data.

vpe_recv(PORT_NAME, index, ...);
     Receive buffer.

vpe_nrecv(PORT_NAME, index, ...);
     Non-blocking receive.

vpe_probe(PORT_NAME, index, ...);
     Test for message arrival.

vpe_precv(PORT_NAME, index, ..., buffer, buflen, type, actual_len);
     Receive and unpack data.

vpe_mcast(PORT_NAME, index_range, ...);
     Send current buffer to multiple node instances.
```

Collective Routines

```
vpe_scatter(data, sendcount, recvcount, type, root);
     Root node instance divides data among all instances.

vpe_gather(data, sendcount, recvcount, type, root);
     Each node instance (including root) sends data to the root.
```

6.0 Building and Running Programs

VPE programs can be built and run directly from the VPE environment. When the user picks one of the build options from the Build menu, VPE will generate C (or Fortran and C) code to implement the visually specified program. It will then begin a parallel and distributed compile of all files produced that will ensure that all modules are built for all necessary machines. Once the compiles terminate successfully, the user can begin a program execution from the VPE environment.

7.0 Animation

VPE will have the ability to animate the execution of a VPE program. This animation will be performed directly on the users VPE programs. It will include highlighting arcs when messages are sent on them. In PVM environments, XPVM [Gei94] will be able to show other animated displays.

8.0 References

[Beg91] 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, pages 435--444, Albuquerque, 1991.

[Beg93] A. Beguelin, J. Dongarra, G. A. Geist, and V. S. Sunderam, "Visualization and Debugging in a Heterogeneous Environment," IEEE Computer, v. 26, no. 6, June, 1993.

[Gei94] 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.

[MPI94] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", Journal of Supercomputing Applications, Vol. 8, No. 3/4, 1994.

[New92] P. Newton and J.C. Browne, "The CODE 2.0 Graphical Parallel Programming Language," Proc. ACM Int. Conf. on Supercomputing, July, 1992.

[New93] 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.

[Ous94] J. Ousterhaut, Tcl and the Tk Toolkit, Addison-Wesley, Reading, MA,1994.