Current software releases and reports regarding HeNCE can be obtained from netlib by sending electronic mail to netlib@ornl.gov containing the line "send index from hence". Instructions on how to receive the various parts of the HeNCE system will be sent by return mail.

References

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

[2] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. A users' guide to PVM parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

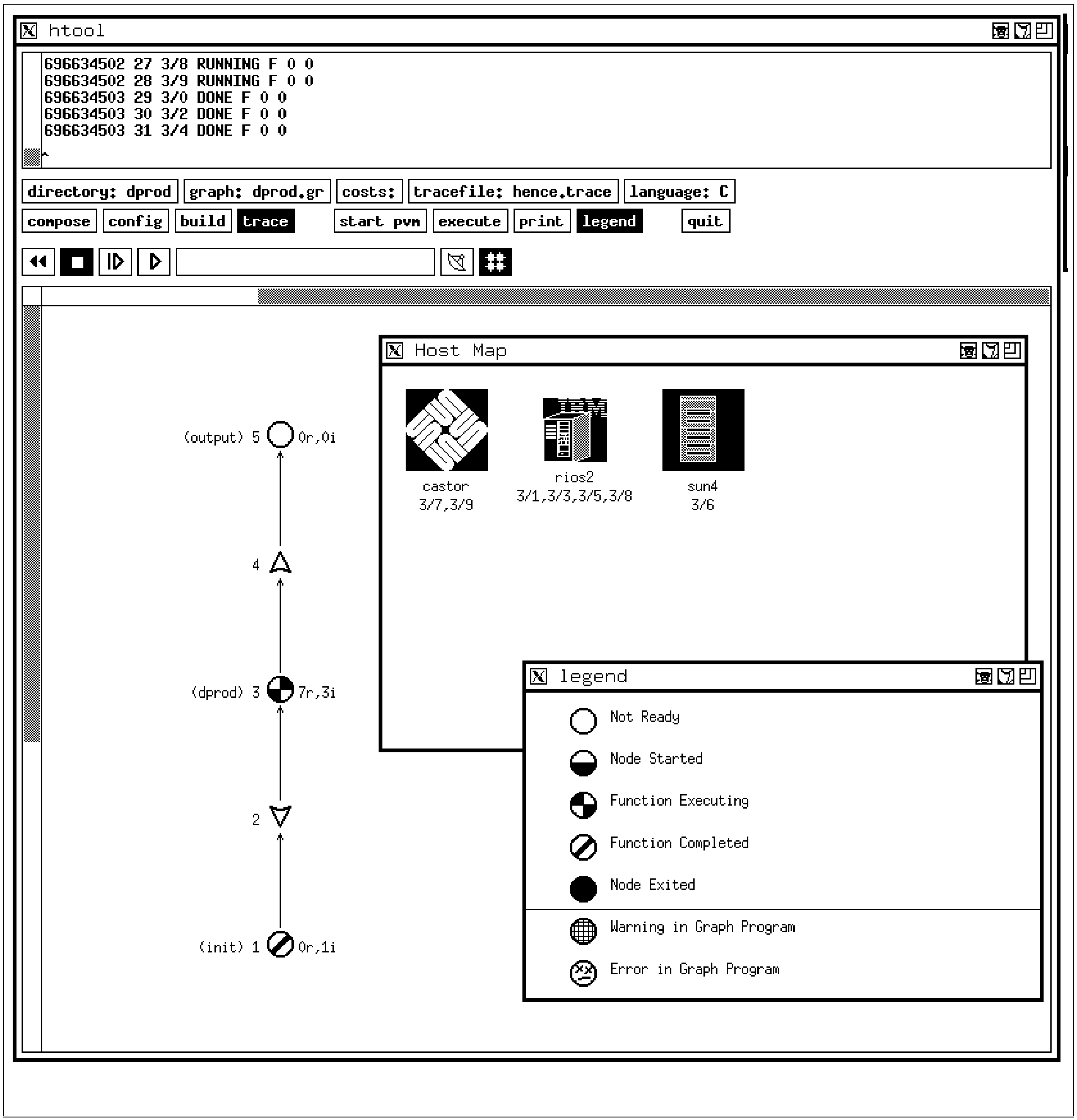

Figure 18: The dot product trace

```
Usage:
  master
    <graph-file>                 { or "-" for standard input }
    [ -cm <cost-matrix-file> ]   { default is cost.mat }
    [ -tf <trace-file> ]         { default (or "-") is stdout. "-2" is stderr}
    [ -nt ]                      { for no tracing }
    [ -trp ]                     { for tracing parameters }
    [ -trm ]                     { for tracing pvm messages }
    [ ? ]                        { To get this message }
```

For most efficient execution the `-nt` option should be used. This option turns off tracing and thus eliminating the tracing overhead.

For the dot product one would simply execute the command `master dprod.gr -cm mycost.mat -nt`. This is assuming the graph is stored in the file `dprod.gr`, the costs are stored in the file `mycost.mat`, and that PVM is running on the local machine plus the machines in the cost matrix.

4.6 Tracing Dot Product

As the dot product program executes, HeNCE creates a file containing trace events that indicate the progress made by the program. Figure 18 shows trace mode for dot product. This trace can be useful for debugging or fine tuning a program's execution. Htool can animate the display of this trace file. To see a trace animation, enter trace mode by pressing the `trace` button. In trace mode you will see a row of buttons and a large window. The four buttons on the far left control the playback of the tracefile. The first button rewinds the file. The second stops playback. The third will single step through the tracefile and the fourth will continuously play through the tracefile. The slider next to these buttons controls the playback speed. The second to last button, the one that looks like a satellite dish, is the monitor button. If this button is pressed before playback, htool will assume that the program is currently executing and will show the trace file up to the current point then wait for new trace events to come in, displaying them as they arrive. The last button brings up a window showing the machines used in the trace mode. These machines are represented by icons with the name of the machine under the icon.

5 Status and availability

The current HeNCE programming tool, htool, is the result of a lot of coding in a relatively short period of time. As a result it is not complete. We solicit feedback from the users. If you have any bugs to report or suggestions to make regarding HeNCE please send them via email to hence@msr.epm.ornl.gov. When you have bug reports, please specify:

- the machine type you are running on (vendor, model),
- the operating system version (ie ULTRIX V4.1),
- the Version of X windows (R4, R5, IBM's...), and
- the window manager you're using.

There are many enhancements that we plan for htool. We plan on revising the node program syntax to make the partitioning of arrays more natural. We envision something along the lines of FortranD syntax for the node programs would be helpful. In the future we would like to support hierarchal HeNCE graphs. The current make facility doesn't help much when multiple architectures are being used. Htool needs to support a remote make facility. This facility would include feedback to the user about which machines need to be updated. Under user direction htool could also copy source and makefiles around the virtual machines.

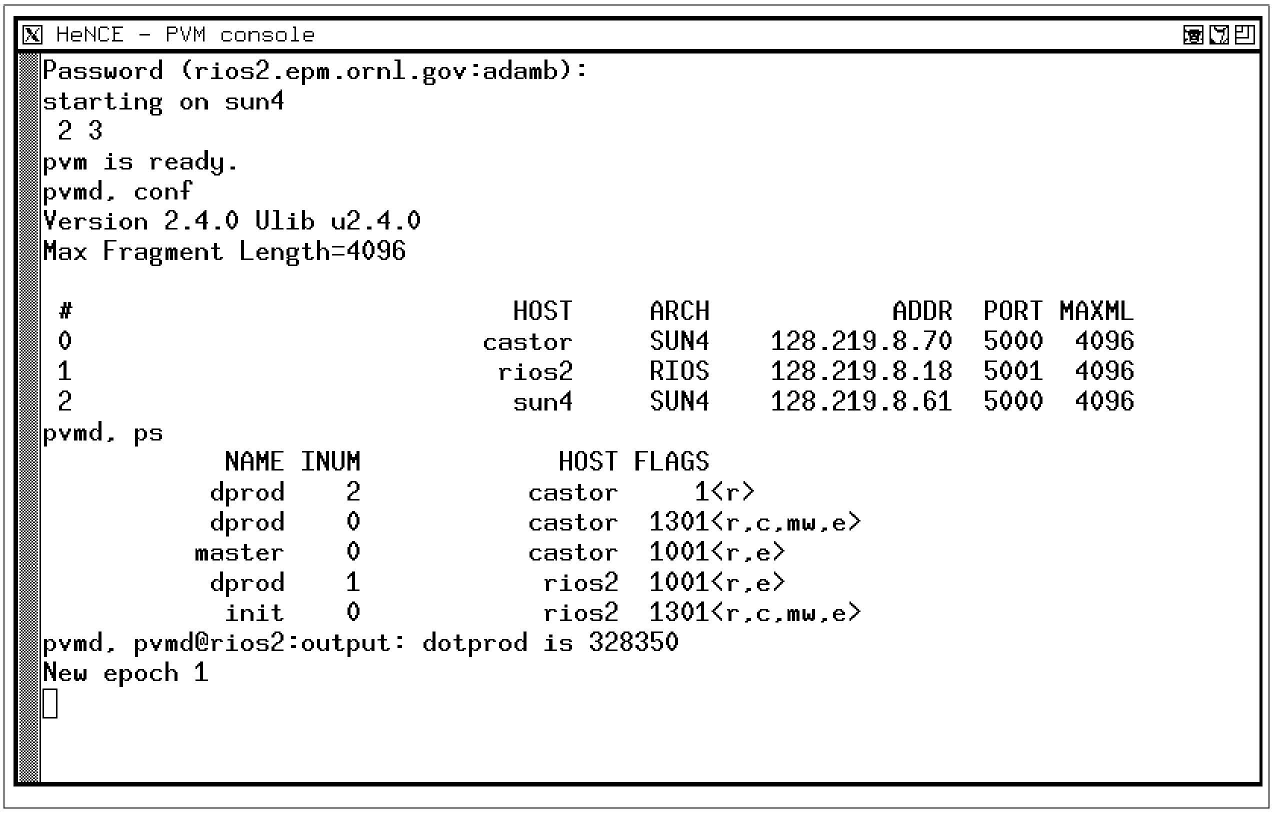

Figure 17: The PVM console after executing dot product.

in adamb's pvm directory rather than the default /tmp/pvm/pvmd. The pvm_hosts file is just a list of machines and defaults to use. If your machines are setup in a standard configuration then you probably will not even need a PVM hosts file. You may use another file name instead of /dev/null by setting the X resource htool.pvmHostFile to something else.

Once you see the message `pvm is ready.` in the PVM console window then you may press the `execute` button and htool will execute your program. Output from prints in your program will show up in the PVM console window. You may do a ps on the PVM console to see where your HeNCE program is executing. Information about your program's execution will appear in htool's message center window.

Figure 17 shows the PVM window after executing the dot product program.

4.5 Executing Dot Product without htool

It is possible to execute the dot product program without using htool. This is useful if an X window server is not available, development of the program is finished, or the program is to be run in batch mode.

The program `master` will execute a HeNCE program. Master needs a graph file and a cost matrix for input. PVM must already be running and the executables for the HeNCE program must be installed in the correct directories. The `master` program takes the following parameters:

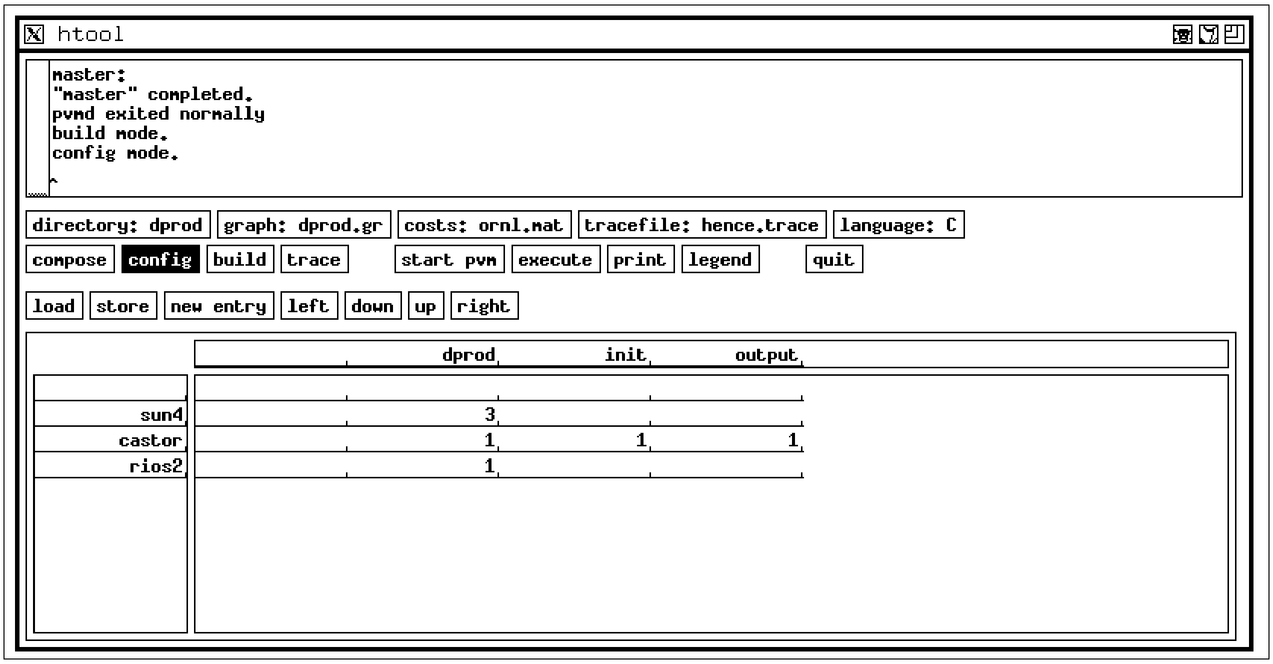

Figure 16: Htool in configure mode.

4.3 Configuring a Virtual Machine for Dot Product

When htool executes the dot product program, it will map nodes in the graph to machines in the virtual machine. This mapping is done based on a cost matrix. The cost matrix is specified in htool's config mode. Enter config by pressing the `config` button on htool's main button panel. A new row of buttons and a matrix widget will appear in the bottom half of the window. (See Figure 16). The left column of the cost matrix is used for machine names. The top row is for node subroutine names. The matrix entries show the cost for running a node on a machine. Costs are positive non-zero integers. The higher the integer, the higher the cost for running the subroutine on that machine. If the cost is blank then the HeNCE will not execute the subroutine on that machine.

Every subroutine must have a cost for at least one machine. In other words, no subroutine may have a completely blank column. For our example we need to input costs for the node subroutines called init, dprod, and output.

4.4 Running Dot Product

Before we can execute the HeNCE program we need to start the virtual machine. This is done by clicking the `start pvm` button. An xterm running the master pvmd will appear. This window is the PVM console. When the pvmd is started it is given a hostfile which has been constructed from the cost matrix and the pvm_hosts file. The cost matrix tells htool which hosts to use. The pvm_hosts file give options for these various hosts. For instance, your cost matrix may have a machine named elvis. The pvm_hosts file may have the line:

```
elvis lo=adamb pw dx=/home/adamb/pvm/SUN4/pvmd
```

The lo option means that when PVM is started on elvis it should use the login name adamb. The pw tells PVM to ask for a password for adamb on elvis. Finally the dx option tells PVM to use the daemon

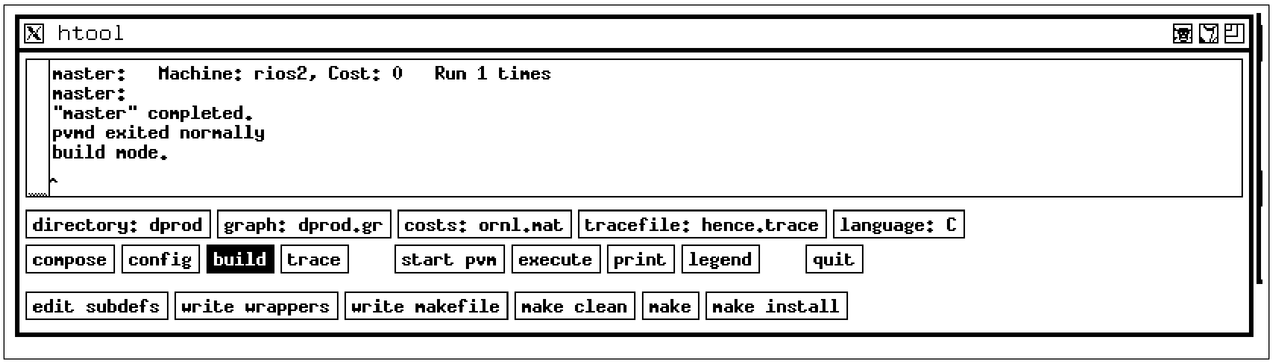

Figure 14: Htool in build mode.

```
void init(float*, float*, int);
void dprod(float*, float*, int, float*);
void output(float*, int);
```

Figure 15: The "subdefs" file for the dot product program.

A makefile can be generated in build mode by clicking **write makefile**. The makefile is customized for the current graph. The location of certain libraries and include files specified in the makefile come from the settings of your X resources. Thus you may want to check the makefile after it is generated to see that the directories are set correctly. The command `xrdb -query | grep htool` will show you the current settings of your htool X resources. If you have special libraries other than PVM that your code needs, you will have to further customize the htool generated makefile. Once the wrappers and the makefile have been generated, click **make clean** then **make install**. The make clean will remove any old object files that may exist in your directory. The make install will compile your code and the wrapper code. These object files will be linked with the HeNCE and PVM libraries and then be placed in the PVM binary directory (usually \$(HOME)/pvm/\$(ARCH)). Error messages from the make will appear in the scrollable message center in htool's top window.

The make from within build mode only compiles on the machine where htool is running. If you have a remote machine (i.e. not the same NFS file system) that you will be using in your virtual machine then you will have to copy the makefile, your source files, and the wrapper files to the remote machine and do a make install there. If the you have another local machine of a different architecture which will be participating in the execution of the program, then you will have to log onto that machine, change to the dprod directory and type `make clean` to remove old object files and `make ARCH=xxxx install` where xxx is right for that machine. This will create the executables for that architecture. It is also possible that you may have different source for different machines. For instance, the dprod node subroutine could be implemented on the CM2 using Paris calls. The wrapper code would be the same for the dprod node but the makefile would need to be changed to include linking to the Paris library.

Now that we have generated executables for our dot product program, we need to tell htool which machines we want to use in our virtual machine.

```
void output(s,n)
float *s;
int n;
{
    float total = 0.0;

    while(n--) {
        total += s[n];
        }

    printf("output: dotprod is %g\n", total);
}
```

Figure 13: Node subroutine for the output node.

but they are typed in using a text editor. The node programs are actually stored as part of the graph. If you change a node program you will need to store the graph to make the changes permanent. This is a bit deceiving since you actually store the node program in a file when you are finished editing. This file is in fact just a temporary file. What you store in the temporary file is immediately parsed and placed into the graph data structure within htool. This means that syntax errors in node programs are immediately detected. The node programs are stored in parsed form within htool. This means that any extra white space or parentheses will be stripped out of your node program. A future version of htool will also store your node program the way you typed it in. The node subroutines are the subroutines that HeNCE will call during run time. HeNCE assumes that you will have one source file for each node subroutine and that the name of the source file will be the same as the name of the node subroutine. Any other subroutines the node subroutine may call should be in the same file. If they are in a separate file, then you will have to alter the makefile automatically generated by htool.

Now that the program has been written, the next step is to create executables for the program.

4.2 Compiling the Dot Product Program

Build mode (Figure 14) supports creation of executables. Several steps are involved in building executables for a HeNCE program. Wrappers for each function in the HeNCE program graph must be written. A wrapper is source code which calls the node subroutine. This wrapper code calls the HeNCE and PVM libraries to obtain the necessary parameters for a subroutine then it calls the node subroutine. Once the node subroutine returns, the wrapper code may spawn any children and send them their parameters. The wrapper code also keeps htool up to date as to the status of its node by sending out trace events.

The **write wrappers** button in build mode will create the wrapper source code for each node in the program graph. These wrappers are generated from definitions is the subdefs file. The subdefs file contains information on the type and number of parameters each routine needs. Currently this file is generated by the user. Clicking **edit subdefs** will allow you to edit the subdefs file. This button also exists in compose mode because it is easer to write a subdef when looking at the actual source code for the routine. Figure 15 shows the subdefs for the dot product example.

The subdefs file is consulted when the **write wrappers** button is pressed. This file can contain more definitions than are used in the graph. Only wrappers for the subroutines in the current graph will be generated. *It is important that the current setting of the language button be correct when the wrappers are written.* Fortran wrappers differ from C wrappers. You should also note that only one wrapper is generated for each subroutine. A node subroutine may be called by several different nodes in the graph but there will still only be one copy of the source code and one wrapper for this subroutine.

```
NODE [ 400 300 ] 3
    NEW < float a[n/nproc]=x[p*(n/nproc):(p+1)*(n/nproc)-1];
    NEW < float b[n/nproc]=y[p*(n/nproc):(p+1)*(n/nproc)-1];
    <> sum[p];
    dprod(a, b, n/nproc, &sum[p]);
```

Figure 10: Node program for the dprod node.

```
void dprod(a,b,n,s)
float *a, *b;
int n;
float *s;
{
    *s = 0;
    while(n--) {
        *s += a[n]*b[n];
        }
}
```

Figure 11: Node subroutine for the dprod node.

to compute the overall dot product from the individual elements of the sum vector.

The source code for the dprod is simply a sequential dot product subroutine. It multiplies the elements of the two vectors and accumulates them in a sum. Figure 11 shows the dprod C source.

The fanin node, node 4, is really just a place holder. It marks the end of the fanout started by node 2. All nodes between nodes 2 and 4 will be replicated at runtime based on the node program in the fanout node. There are no real node programs or node subroutines associated with fanin nodes.

The output node, node 5, sums up the elements of the sum vector and prints the results. As Figure 12 shows, the node program for the output node has two input variables that it uses. The brackets on the sum variable indicate that the entire sum array will be used by this node.

The C code for output is shown in Figure 13. The output routine simply totals up the sum array and prints this total. Since these routines will be running under PVM any of their output will go to the PVM console window.

Now we have created a HeNCE program for computing a dot product. There are three main parts to a HeNCE program: the graph, the node programs, and the node subroutines. The graph is created by clicking the mouse in compose mode to draw nodes and arcs. The help button displays a synopsis of what each mouse click does. The node programs describe the data that a subroutine will need, how that data is to be partitioned, and how the subroutine is to be called. Node programs are also entered in compose mode,

```
NODE [ 400 100 ] 5
    < float sum[ ];
    < int nproc;
    output(sum, nproc);
```

Figure 12: Node program for the output node.

```
NODE [ 400 500 ] 1
    NEW <> int nproc=10;
    NEW <> int n=100;
    NEW <> float sum[nproc];
    NEW <> float y[n];
    NEW <> float x[n];
    init(x, y, n);
```

Figure 7: Node program for init node.

```
void init(x,y,n)
float *x, *y;
int n;
{
    printf("init\n");
    while(n--) {
        x[n] = n;
        y[n] = n;
        }
}
```

Figure 8: Node subroutine for init node.

button on node 1. Htool will then start an edit session with the file init.c. If the language switch is set to Fortran then an edit session for init.f will be started.

The fanout node, node 2, has no node subroutine attached but it does have a node program. Figure 9 shows the simple node program for this fanout node. This fanout node program indicates there will be nproc subgraphs created and they will be numbered from 0 to $nproc-1$. At runtime nproc copies of dprod will be started, each will have access to its own copy of the variable p indicating its instance number. In this sense dprod is like the body of a loop, it is executed nproc times with $p = 0 \dots nproc-1$, except that all of these invocations of dprod are going on in parallel.

The node program for dprod, Figure 10, declares two new variables, the vectors a and b. These variables are initializes with sub-vectors of x and y respectively. The subrange is defined by the expression `p*(n/nproc):(p+1)*(n/nproc)-1` . The expression on the left side of the colon is the starting index and the expression on the right side is the ending index. The vector a will be allocated and initialized to this range of x. The same range of y will be assigned to b. The range expression will work as long as nproc divides n evenly. A more complicated expression is needed in the general case.

The declaration `<> sum[p]` means that a single element of sum is going to be used in this node. Specifically, the p^{th} element of sum is sent, by address, to the dprod function. This allows each parallel copy of dprod to write to a single element of the sum array. The entire sum vector is later used by the output node

```
FANOUT [ 400 400 ] 2
    p = 0 TO nproc-1;
```

Figure 9: Node program for the fanout node.

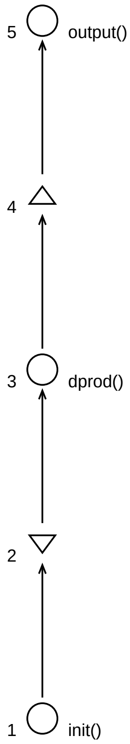

Figure 6: HeNCE graph for dot product example.

this program is shown is Figure 6. In this program the vectors are split into contiguous sub-vectors and the dot product is computed for each sub-vector in parallel. The sums from each sub-vector are collected and added together to produce the final output.

4.1 Composing Dot Product

In order to create the dot product program you will need to go into compose mode. Enter compose node by clicking on the `compose` button in the htool window. First you should draw the program graph. The help button in compose mode will give you the details of which buttons to use when drawing.

The first node, init, declares and initialized the variables for the computation. The node program is shown in Figure 7. Here the variable nproc is declared as a new integer and initialized to 10. This is the number of processes that will be created by node 2. The length of the sum vector is also nproc. Each entry in the sum array will be computed by a separate process. The variable n is declared and set to 100. The variable n is the length of the x and y vectors. Finally, the init subroutine is called, passing x, y, and n.

The C source for init can be seen in Figure 8. Init simply initializes the vectors x and y such that $x_n = n$ and $y_n = n$.

The node program and the node subroutine are entered by the user in compose mode. Clicking the right button on node 1 will bring up an edit session for the node program associated with node 1. The first line of the node program, in this case `NODE [ 400 500 ] 1`, is generated by htool. `NODE` indicates the type of node, in this case a normal node. The numbers in brackets are coordinates of the node's location. The integer, 1 in this case, is the node id. To edit the source code, init.c, hold the shift key and click the right

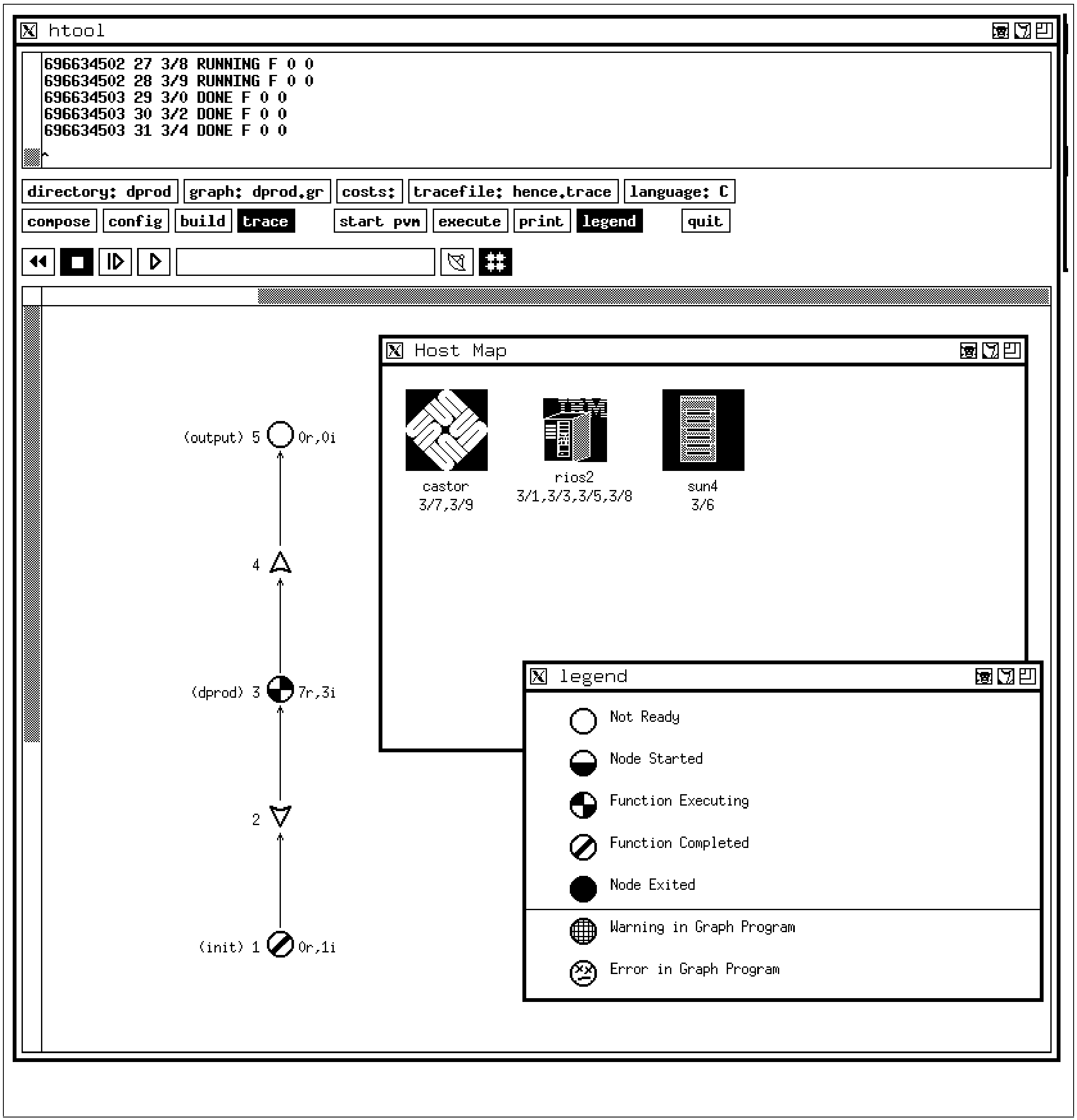

Figure 5: The dot product trace

If, however, it expects *m1* and *m2* to be large, one-dimensional chunks of bytes, and it will produce p that way, then the wrapper should be defined as:

```
void mmult(int *, int *, int **, int, int, int);
```

The rule of thumb to use here is: If an array is to be passed as a 1-d chunk of bytes, then the wrapper should define it as an `(int *)` (or `(double *)` or whatever type is desired). Otherwise, there should be as many `*`'s as there are dimensions in the array. If the address of a chunk of bytes is to be passed, then the wrapper should define an `(int **)` (or whatever type is desired). Otherwise, there should be as many `*`'s as there are dimensions, plus one.

HeNCE will not convert array types to wrappers. Thus, if the node declares

```
NODE 0
  <> int m1[15];
  proc(m1);
```

and the wrapper for *proc()* is:

```
void proc(double *);
```

HeNCE will flag an error.

3.5 Tracing a Program Run

As a HeNCE program runs it generates trace events. These events can be used to drive an animation of the HeNCE program either during or after its execution.

Before entering trace mode the program graph to be traced should be loaded. This can be done either by using the `graph:` button on the top line of the command panel or by using the `load` button in compose mode. To view a program trace enter trace mode of htool by clicking on trace. The current program graph will be displayed. See Figure 5. The row of buttons just above the horizontal scroll bar are used to control the playback of the trace animation. The first button is the rewind button. It rewinds the trace file to its beginning. The second button is the stop button. It can be used to stop playback of the tracefile. The third button is used for single stepping through the tracefile. The fourth button is the play button, used to continuously play back the tracefile. Next to the play button is a slider. This slider indicates the speed of the trace playback. Clicking the middle mouse button on the left end of the slider will slow down the playback. Clicking on the right end of the slider speeds up the playback. The second button from the right (the one that looks like a satellite dish) is the monitor button. This indicates that the HeNCE program is currently executing and that the trace tool should wait for new trace events to arrive. The last button will display the network of machines used to execute the HeNCE program.

During the trace, nodes in the program graph and the machines in the network display are highlighted to indicate their state. The various colorings are explained by the legend. The legend can be displayed by clicking on the `legend` button.

3.6 Printing from HeNCE

It is possible to output a printable copy of the HeNCE graph. The `print` button in htool will generate a postscript version of the HeNCE graph. You can print this the graph on your printer by using the command `lpr graph.ps` where graph.ps is the filename you selected from the print window.

4 An Example: Dot Product

In this section we will create a simple parallel dot product program from start to finish. The dot product program will initialize two vectors and compute the dot product of these vectors in parallel. The graph for

2. **Executing a subroutine**: The computation of a node gets performed by a C or Fortran subroutine. The arguments to the subroutine can be constants, parameters, or expressions involving parameters. The return value of the subroutine can likewise be bound to a parameter, and through pointer indirection, the subroutine can modify parameters if they are properly given as arguments.

3. **Sending parameters**: When the subroutine is over, the node's execution is considered over, and its children can be started. The node cannot go away, however, as its children and other descendents may need its parameters. Thus, it stays alive to service requests for parameters. When it is sure that no more descendents will ask for parameters, it may go away.

Wrappers In order to have HeNCE correctly call a subroutine, a *wrapper* must be generated. This wrapper is linked with the subroutine and the HeNCE libraries to make an executable which will run with HeNCE tools on PVM.

For example, we'll use a subroutine *add()*, which adds two `double`'s and returns their sum. To make a wrapper for *add()*, we use the program `mkwrap`, which takes descriptions of subroutines and makes wrappers for them. The description of a subroutine must be of the form:

```
return_type sub_name '(' arg1_type ',' arg2_type> ',' ...  argn_type ')' ';'
```

If there is no return value, then *return_type* should be `void`. The description for *add()* should be:

```
double add(double, double);
```

Mkwrap takes that description, and produces `w_add.c`, which should be compiled and linked with `add.c` and the HeNCE libraries to make the executable `add`. `Add` should be put into `~/pvm/$ARCH`, so that PVM can find it.

As before, HeNCE attempts to do proper type conversion, so if *add()* is passed two `int`s is to return an `int`, it will convert the arguments to `double`s, call *add()*, and convert the return value back to an `int`.

Arrays and output arguments can be tricky – the following examples should be illustrative: Suppose we have the routine *rand()* which returns a random number in its first argument. An example HeNCE node using *rand()* would be:

```
NODE 0
  NEW int a;
  rand(&a);
```

and the wrapper definition should be:

```
void rand(int *);
```

Suppose now that you have a routine *mmult()* which takes two matrices and their sizes, and allocates a product matrix, which it returns in its third argument. An example HeNCE node using *mmult()* would be:

```
NODE 0
  <> int m1[r1][c1];
  <> int m2[r2][c2];
  mmult(m1, m2, &p, r1, c1, c2);
  > int p[r1][c2]
```

(note that the declarations of *m1* and *m2* are unnecessary – they were just added for clarity). If *mmult()* expects to receive an `(int **)` for *m1* and *m2*, and produce an `(int **)` for *p*, then the wrapper definition for *mmult()* should be:

```
void mmult(int **, int **, int ***, int, int, int);
```

3.3 Configuring a Parallel Virtual Machine

In order to execute a HeNCE program the user must specify which machines are to be used as a virtual machine. This is done in the config mode of htool. Config mode presents the user with a spreadsheet like interface as shown in Figure 16. The lower window in config mode represents a HeNCE cost matrix. The left column is intended for machine names such as betty.cs.utk.edu. The top row of the spreadsheet is for node subroutine names. Entries in the interior of the matrix represent the cost of executing a node subroutine on a particular machine. Each entry is a positive integer; the higher the number the higher the cost of running a subroutine on that machine. The cost is a user defined parameter. HeNCE does not attempt to estimate this cost, that is the job of the programmer. Entries in the cost matrix are made by highlighting an entry with the mouse and typing in the value for that entry. If a new row or column is needed, simply click on the `new entry` button.

Cost matrix files can be loaded and stored using the `load` and `store` buttons respectively.

3.4 Executing a Program Graph

Before executing a HeNCE program the three following things must have been done:

- Load a graph file
- Load a cost matrix
- Compile and install binaries

Once these three steps have been completed, click the `start pvm` button. At this point htool will create a parallel virtual machine. The hosts used to create the virtual machine are taken from the cost matrix. A PVM console window will appear. PVM console commands may be issued in this window. For instance, typing "conf" at the pvmd prompt will show which machines are currently configured in this virtual machine. Typing "help" will display a list of commands which are available for the pvmd. See [2] for more details on using PVM.

Once the message "pvm ready" appears in the console window the htool can execute the HeNCE program. To execute the program on the current PVM, click on the `execute` button. At this point htool will execute the program. Information about the execution will appear in the htool information window.

Any output statements from HeNCE node subroutines will appear in the PVM console window. During the execution, a file containing trace information is created. The name of this file is "hence.trace" in the current directory.

3.4.1 What gets done in a node

Each node in a HeNCE graph is a distinct process on some machine. The arcs of the graph specify when a node gets executed, which is when all of its parent nodes have finished executing. The nodes communicate to each other by *parameters*, which are specified in the *node-program*. Basically each node has three phases of its execution:

1. **Getting parameters**: The first thing that a node does is read in its specified parameters. It first tries to get the parameters from its parent nodes. For example, if a node's parent specifies that it has parameter B, and the node needs parameter B, then the node sends a request for B to the parent. The parent services that request by sending the value of B to the node. If no parents have a parameter, then the grandparents are checked, and so on.

   Certain types of parameters, like arrays, can be split up among nodes. If a node wants all of an array and its parents have split it up, then it rebuilds the array from its parents. If the parents have just split up part of the array, then it gets what it can from them, and tries to get the rest from its grandparents, and so on.

```
NODE 0
  NEW <> char a[3] = "abc" ;
  NEW <> int b[3] = a;
  NEW <> float c[3] = b;
  NEW <> int d = 3;
  NEW <> double e = d;
  null()
```

b will be{`97, 98, 99`}, c will be{`97.0, 98.0, 99.0`} and e will be 3.0.

3.2 Building an Executable

Build mode supports the creation of executables from the node subroutines. In build mode the user creates a subdefs file which is used to write wrappers for all of the node subroutines. A makefile can be generated in build mode. This makefile is used to compile the node program and its wrapper. Various make commands can be invoked from build.

3.2.1 Generating Wrappers

Build will write source code for each node subroutine in the HeNCE graph. This source code is in the form of programs called wrappers. They are called wrappers because they are meant to wrap around the user supplied subroutine. The wrapper code does the work of gathering up the parameters necessary to invoke a the node subroutine. (The wrapper code makes calls to the PVM system. The programmer never actually writes any PVM code directly.) Wrappers are generated from a file called the subdefs file. This file contains a description of what type of variables a subroutine expects and what type of value, if any, the function returns. A subdefs file might contain the following:

```
void start( float*, int* );
float barray( float*, int*, int* );
void comb( int*, float* );
```

The node subroutines `comb` and `start` do not return a value. The `barray` node subroutine returns a float. The parameters to the `start` subroutine are pointers to a float and an int. In this case the subroutines are probably written in Fortran since all the parameters are pointers.

Clicking on the `edit subdefs` button will pop up a window allowing you to edit the subdefs file. We anticipate eliminating the edit subdefs step in future versions of HeNCE since we should be able to derive equivalent information from the HeNCE graph and its associated node programs.

Clicking on `write wrappers` tells htool to create the wrapper source code. This will create one file for each node subroutine described in the subdefs file. The source files generated will be named cw_subr.c if the language switch is set to C and fw_subr.c if the language switch is set to Fortran. For the example subdefs file described above three files will be generated: fw_start.c, fw_barray.c, and fw_comb.c.

The `write makefile` button will create a makefile for compiling the wrappers and the user supplied node subroutines. If a makefile already exists, htool will ask you to confirm any overwriting of the existing makefile.

The `make clean` button will invoke make with the clean option. Make clean will remove any object files and node subroutine executables in the current directory.

Clicking `make` will attempt to compile the subroutine wrappers and the node subroutines. If a subroutine is named subr, the executable file will be named subr also.

The `make install` button also compiles the subroutines if necessary but moves the executables to the PVM binary directory so they can be executed under the PVM system.

Feedback from any invocations of make will be displayed in the htool's message window.

Loop Nodes Loop nodes are like fanout/fanin nodes, except that the copies of the subgraph are performed serially. The syntax is like the *for* syntax in C:

```
'BEGINLOOP' [ <x location> <y location> ] id
    '(' [ var '=' expression ] ';'
          expression          ';'
        [ var '=' expression ] ')' ;
'ENDLOOP' [ <x location> <y location> ] id ';'
```

when the loop is first encountered, an output integer parameter (*var*) is created, and set to the first *expression* . If the middle *expression* evaluates to non-zero, then the subgraph inside the loop is expanded and executed. When it finishes, then *var* is updated using the last *expression* , and the second expression is rechecked. If it evaluates to zero, then execution continues from the ENDLOOP. Otherwise, the subgraph is expanded and evaluated again. And so on.

A current restriction enforced by HeNCE is that the vars in the first and third expression must be the same. If the vars are not the same, the parser emits an error.

Pipe Nodes Pipe nodes allow control flow to be pipelined through a subgraph. The syntax is similar to fans:

```
'BEGINPIPE' [ <x location> <y location> ] id
    var '=' expression 'TO' expression ';'
'ENDPIPE' [ <x location> <y location> ] id ';'
```

Again, for each value of *var*, a separate copy of the subgraph is spawned. However, now, for each node in the subgraph, there is an arc from its instance in one copy of the subgraph to its instance in the next copy of the subgraph (i.e. from the instance where $var = i$ to $var = i + 1$.

Thus, pipes allow for a more restricted type of parallelism than fanout/fanin nodes. Specifically, if node instances need to be serialized in a certain algorithm, then the pipe construct can be used effectively.

More on Array Passing The internal representation of arrays has two ways of storing them: As a big 1-dimensional chunk of bytes (which is what you'd get in C if you declared an array with constant sizes: e.g.`int a[4][5];` gives you a pointer to a chunk of 20 ints), and as a tree of pointers (which is what you get in C otherwise – e.g.`char **argv`). When HeNCE allocates an array, it allocates the chunk of bytes, and then builds the pointer-tree on top of it, so either representation can be used by the programmer (see Section 3.2.1). Currently, HeNCE stores arrays in row major. For Fortran use, HeNCE will need some modification.

A by-product of HeNCE's storing of arrays is that one can initialize 1-d arrays from n-dimensional ones, and vice-versa. For example, the declaration:

```
NODE 0
  NEW <> char a[10] = "abcdefghij";
  NEW <> char b[2][5] = a;
  NEW <> char c[2][5] = "abcdefghij";
  <> char a[0 : 4];
  null();
```

is perfectly legal. Here b and c will both equal{`"abcde", "fghij"`}. Also, when HeNCE initializes such arrays, it makes them all point to separate chunks of memory. Therefore, changes to a won't affect b, even though the syntax might look that way. Moreover, changes to a won't affect *a[0 : 4]* , as they both point to different chunks of memory.

HeNCE attempts to perform type conversion as much as possible. Therefore, in the following declaration:

```
NODE 0
  readtext(&len, &txt);
  > int len;
  > char txt[len];
```

Output declarations can also be used for parameters that need to be created, but aren't used in the subroutine call. For example, suppose that the array P is going to be distributed among many nodes, but no one node is ever going to hold P in its entirety. P must be declared by some node, so that HeNCE knows its type and size; however, if it is declared as an input or input/output parameter, then HeNCE will allocate it and set its values to null. If it is declared as an output parameter, then HeNCE does not allocate it, but does note its type and size. Therefore the user can use output declarations to declare, but not allocate or use, an array.

Array parameters cannot change size during HeNCE's execution (unless a new parameter is created using `NEW`). For example, if we made the following declaration:

```
NODE 0
  NEW <> int len = 400;
  <> char txt[len];
  readtext(&len, &txt);
```

and *readtxt* changes *len* to 300, *txt* will not undergo a size change – it will remain 400 characters long.

Special-Purpose Nodes Below is a description of the special-purpose nodes and their textual syntax as generated by HeNCE. As stated before, special-purpose nodes all come in pairs surrounding a subgraph. In reality, special-purpose nodes aren't nodes at all, but specify graph rewriting rules. The nature of the special-purpose node, and the values of runtime variables determine how the subgraph gets written. For example, if a conditional node evaluates to false, then the subgraph does not get included in the graph's execution.

Conditional Nodes Conditional nodes allow the user to choose whether or not subgraphs get executed, according to the runtime conditions of parameters. Like all other special-purpose nodes, conditionals come in begin/end pairs. The nodes between the pairs must form a subgraph. Otherwise, HeNCE will flag an error before performing any execution.

The syntax of conditional nodes are as follows:

```
'BEGINSWITCH' [ <x location> <y location> ] id '(' expression ')' ';'
'ENDSWITCH' [ <x location> <y location> ] id ';'
```

The expression is like any expression discussed above. It may contain constants or parameters. All parameters are defaulted. Like C, if the expression evaluates to 0, then it is treated as false, and the enclosed subgraph is ignored. If it evaluates to true, then the subgraph is executed normally.

Fanout/fanin Nodes Fanout nodes allow the user to dynamically create copies of a subgraph which will execute in parallel. Fanin nodes mark the end of the subgraph to be spawned. Their syntax is as follows:

```
'FANOUT' [ <x location> <y location> ] id ';'
    var '=' expression 'TO' expression ';'
'FANIN' [ <x location> <y location> ] id ';'
```

The two *expression*'s denote a range of values for an *var*, which it treated as an output integer parameter. For each value of *var*, a separate copy of the subgraph is spawned. That copy of the subgraph may access its value of *var* as an integer parameter. If the first *expression* evaluates to a value less than the second, then no copys are spawned; execution continues from the fanin node.

```
NEW <> int b = c[5];
NEW <> d[1 : c[5]];
FANOUT 1 i = c[5] TO 10;
foo(d[c[5]]);
```

however, the following are *legal*:

```
NEW <> int b[5] = c[5 : 9];
foo(c[5] + c[6]);
```

The Subroutine Call Each node is bound to a subroutine. The subroutine call is where the arguments to the subroutine are defined, and the return value is bound. The subroutine call is a part of the node program. The syntax of the subroutine call is:

```
[ param = ] sub-name ( [ expression [ , expression ]* ] ) ;
```

For example, the following are subroutine calls:

```
foo();
a = add(b, c);
update_array(&(A[r][c]));
```

The first example simply calls the subroutine *foo*. The second calls the subroutine *add* with the parameters b and c as arguments, and binds the return value to a. The last example calls *update_array* with the address of element (r, c) of array A.

Parameters may be updated by subroutines in three ways:

1. By binding it to the return parameter: e.g. $a =$ foo().
2. By sending its address as an argument: e.g. foo(&a).
3. By passing an array as an argument: e.g. foo(a[]).

If the parameter to which the return value is bound has not been declared, then it is defaulted to be output-only.

Output Declarations There may be some parameters which have been created or initialized by the subroutine call. These parameters can be declared as output-only after the subroutine call. Output declarations are of the form:

```
> type param-name ;
```

Thus, if node 1 wants to create the parameter a and initialize it to the result of *foo*, then it can be declared as follows:

```
NODE 1
  a = foo();
  > int a;
```

A major difference between output and I/O parameters is in the assumptions about allocation: If an array is declared to be an I/O parameter, then HeNCE will allocate the space for it, and initialize it from its ancestors, if any contain the array. If it is declared as output-only, however, HeNCE assumes that the subroutine will allocate the array. Moreover, the size of the output array is determined after the subroutine call, as those determining parameters might be output-only as well.

(Note: This behavior is not fully debugged in the current release of htool.) For example, suppose the subroutine *readtext* reads a bunch of text from a file, and returns the length in its first parameter, and the text in its second parameter. Then the HeNCE declaration for *readtext* should be:

```
NEW < int a = b;
NEW < int b = a;
```

will be flagged as an error.

Arrays Arrays are declared as in C. Each dimension's specification is enclosed in square brackets. These specifications may be expressions involving other parameters. When an array is first declared, it must be with a type. This defines the size of the array, which must remain constant throughout the lifetime of the computation. If a node redeclares an array that an ancestor has already declared, then it is an error if the declarations differ in type or in the size of any dimension.

After an array has been defined, nodes may use rectangular portions of it. Thus, nodes can split up an array to perform parallel computations. In the simplest case, a node can specify a single element of an array. For example, if node 1 wants to use element $(5, 6)$ of integer array p, then it can declare:

```
<> p[5][6];
```

or simply use it:

```
foo(p[5][6]);
```

Larger, rectangular portions of arrays can also be specified by giving the dimension's low and high values. An empty dimension will default to the entire contents of the dimension. The following are examples of specifying portions of a two-dimensional array $A[20][20]$:

- A: This specifies all of A.
- $A[][]$: This also specifies all of A.
- $A[0:5][10:15]$: This specifies the rectangular portion of A starting with $A[0][10]$ and ending with $A[5][15]$. If this is used in an expression, for example:

  ```
  NEW <> int B[6][6] = A[0 : 5][10 : 15];
  ```

  then B is created and its values are copied from that rectangular portion of A. $B[0][0]$ will be equal to $A[0][10]$, and $B[5][5]$ will be equal to $A[5][15]$.
- $A[0:5][11]$: This specifies a 6-element one-dimensional array whose first value is $A[0][11]$, and whose last value is $A[5][11]$.
- $A[11][]$: This is a 20-element one dimensional array whose first value is $A[11][0]$, and whose last value is $A[11][19]$.
- $A[11:11][0:5]$: This is a 1x6, two dimensional array whose first value is $A[11][0]$, and whose last value is $A[11][5]$.

One can use these array specifications in expressions, and one can legally specify different portions of the same array. For example, the declaration:

```
NEW <> int b[6][6] = A[0 : 5][10 : 15];
NEW <> int c[6][6] = A[1 : 6][10 : 15];
```

are perfectly legal. Each portion of A is considered to be a different I/O parameter. This could be the source of nondeterminacy problems if the partitions of A overlap.

The current implementation of HeNCE has been written so that some expressions involving arrays are illegal. In particular, scalar parameters can not be set to array values. The reason for this has to do with implementation details and future versions of HeNCE should remove these restrictions. The following are *illegal* expressions:

a null value. The following are examples:

```
< a;
< int b = 5;
< char c[25] = "Foo";
< int d[10][15];
```

The first example states to load a from ancestors. The type of a is inherited from the ancestor's declarations. If a doesn't exist, then HeNCE will flag an error. The second example says to load the integer b from ancestors, and if it doesn't exist in an ancestor, initialize it to 5. The third example is similar, except c is an array of 25 *char*'s. If no ancestor has c, then 25 chars are allocated, and the first four are initialized to "foo". Finally, the last example retrieves the two-dimensional integer array d from ancestors. If no ancestor has d, then it is allocated, and all its elements are set to zero.

This is a bit counter-intuitive, since writing `< int b = 5;` doesn't mean that b will be initialized to 5 (in other words, if the node's parent sets b to 10, then b will be 10, and not 5). Thus, there is an alternate input declaration form:

```
NEW '<' type param-name [ '=' expression ] ';'
```

which means that the parameter will be created anew here. Therefore, if a node specifies `NEW < int b = 5;` , then b will indeed be initialized to 5 in this node, regardless of its value (or type) in ancestor nodes.

Input parameters are for input only. They will not get sent to descendents. Thus, when a node declares a parameter for input, it is free to modify that parameter and those modifications will not be seen by any descendents.

If a node wants parameters to be loaded from ancestors and to be seen by descendents, then it can declare them to be Input/Output. This is done by substituting '<>' for '<' in the above syntax. Thus the following examples:

```
<> a;
NEW <> int b = 5;
<> char c[25] = "Foo";
<> int d[10][15];
```

mean the same as above, except that after their subroutine completes, their values can be sent to descendent nodes.

Expressions and default parameters Previously we stated that parameters may be initialized to *expressions*. Expressions may also be used to define array dimensions and subroutine arguments. An expression is defined as a combination of constants, parameters and C operators. Thus, the following are expressions:

```
5
a
b[3][5] + c * 5.0
(d != 0) ? &d : &e
```

If an expression uses a parameter that is not declared, then that parameter is defaulted to be an input/output parameter. Thus, in the declaration:

```
<> a;
< int b = a;
```

the first line can be omitted: Since b mentions a in its expression, a is defaulted to an input/output parameter. Even if an expression is not used (e.g. b is loaded from an ancestor, and is never initialized to a), the undeclared parameters that it uses are still defaulted to input/output parameters.

Circular references are not allowed. For example:

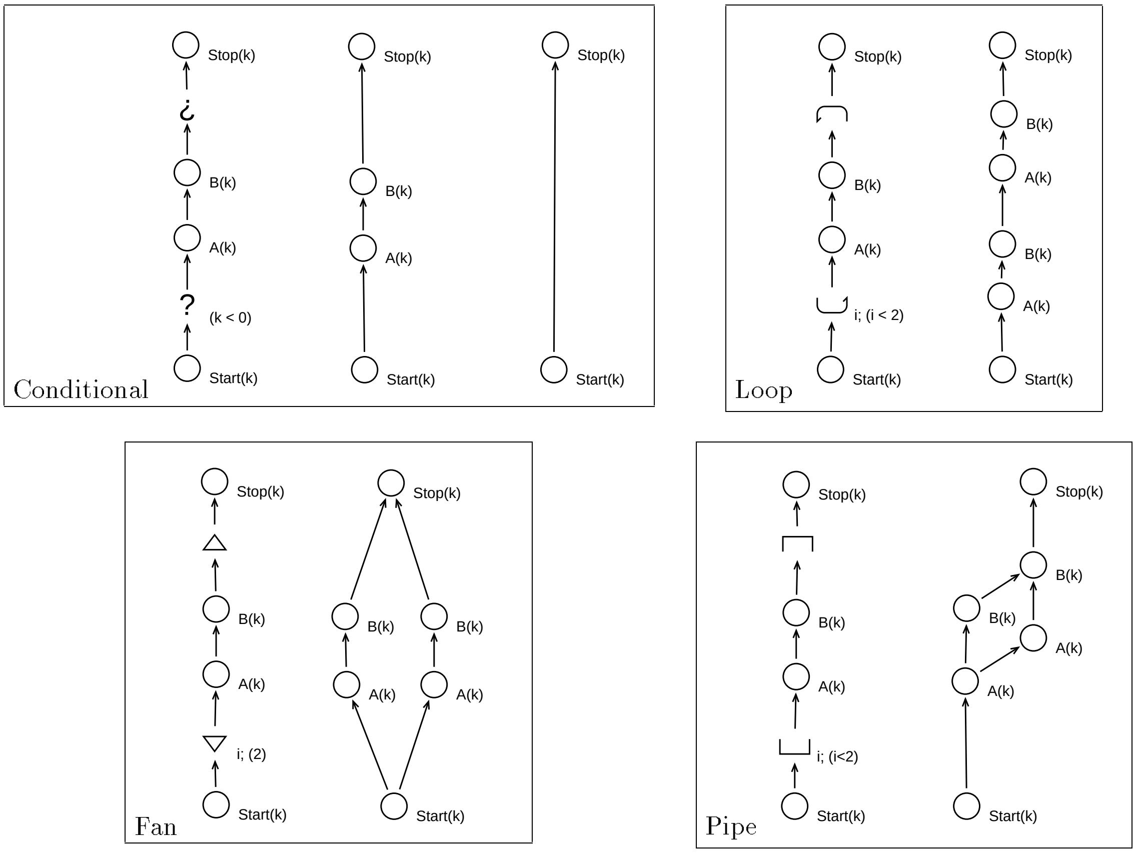

Figure 4: HeNCE conditional, loop, fan, and pipe graph constructs.

They state that a node is to find and load the variable *param-name* from its ancestors. If a *type* is not specified, then the parameter is assumed to exist in ancestor nodes, and it will retain the type declared there. If a *type* is specified and disagrees with the type declared by an ancestor, then HeNCE will flag an error. If no ancestors contain the parameter, then it is created and initialized to *expression* . If there is a type but no ancestors contain that variable and no *expression* is given, then it is created and initialized to

Mouse Button		Shift	Control
Left	Add Node	Move Node	Delete Node
Middle	Add Arc	Move Arc	Delete Arc
Right	Edit Node Program	Edit Node Subroutine	

Table 2: Default mouse actions in compose mode.

node programs see Section 3.1.3.) The node program is considered a part of the graph and is stored along with the graph. *Thus changes to the node program are not saved until the graph is stored.* To edit the node subroutine (the actual C or Fortran source code) for a node, hold the shift key while clicking the right button near the node. HeNCE will start an edit session for that source file. The filename is the name of the function appended with a `.c` or `.f` based on the current setting of the language switch.

3.1.2 Special-purpose Nodes

HeNCE provides constructs which denote four different types of control flow: conditionals, loops, fans, and pipes. These constructs allow the user to specify dynamic control flow, which depends on parameters at runtime. The constructs are defined by special purpose the nodes shown in the lower row of Figure 3. There is a begin and end node for each construct. For instance, there are begin-loop and end-loop nodes for defining a subgraph making up a loop body. The special-purpose nodes all come in pairs, and must *contain* a HeNCE subgraph. In simple terms constructs can be nested but not interlaced.

The four constructs can be thought of as graph rewriting primitives. Figure 4 illustrates these four constructs, showing each construct and an equivalent version of the graph. These constructs add subgraphs to the current program graph based upon expressions which are evaluated at runtime. Using the conditional construct the programmer may specify a subgraph to be executed conditionally. If the boolean expression attached to the begin-conditional node evaluates to true then the subgraph contained between the begin- and end-conditional nodes is added to the program graph. If the expression evaluates to false then the contained subgraph is not added. The loop construct is similar to the conditional in that it specifies a subgraph to be conditionally executed. However, the loop construct also allows iteration on a subgraph as a loop body. In other words, the subgraph making up the loop body is repeatedly added to the program graph based upon a boolean expression that is evaluated each time through the loop. The fan construct in HeNCE allows the programmer to specify a parallel fanning out and in of control flow to a dynamically created number of subgraphs. The integer expression attached to the begin-fan node is evaluated to determine how many subgraphs will be created. Each subgraph created by the fan construct executes in parallel. The pipe construct in HeNCE provides for pipelined execution of a subgraph. The expression attached to the begin-pipe node indicates whether another data item is to be piped though the subgraph. If the expression evaluates to true then another subgraph is added to the graph in order to execute the additional data item in a pipelined fashion. Thinking of these constructs as graph rewriting primitives not only provides a mechanism for specifying parallelism but also a natural way of viewing the dynamic parallelism as a graph unfolds at runtime.

3.1.3 HeNCE Node Language

Node programs consist of three parts: **Input declarations**, the **subroutine call**, and **Output declarations**, which correspond to three node stages: getting input parameters, executing node subroutine, sending output parameters.

Input declarations Input declarations are of the form:

```
< [type] param-name [ = expression ] ;
```

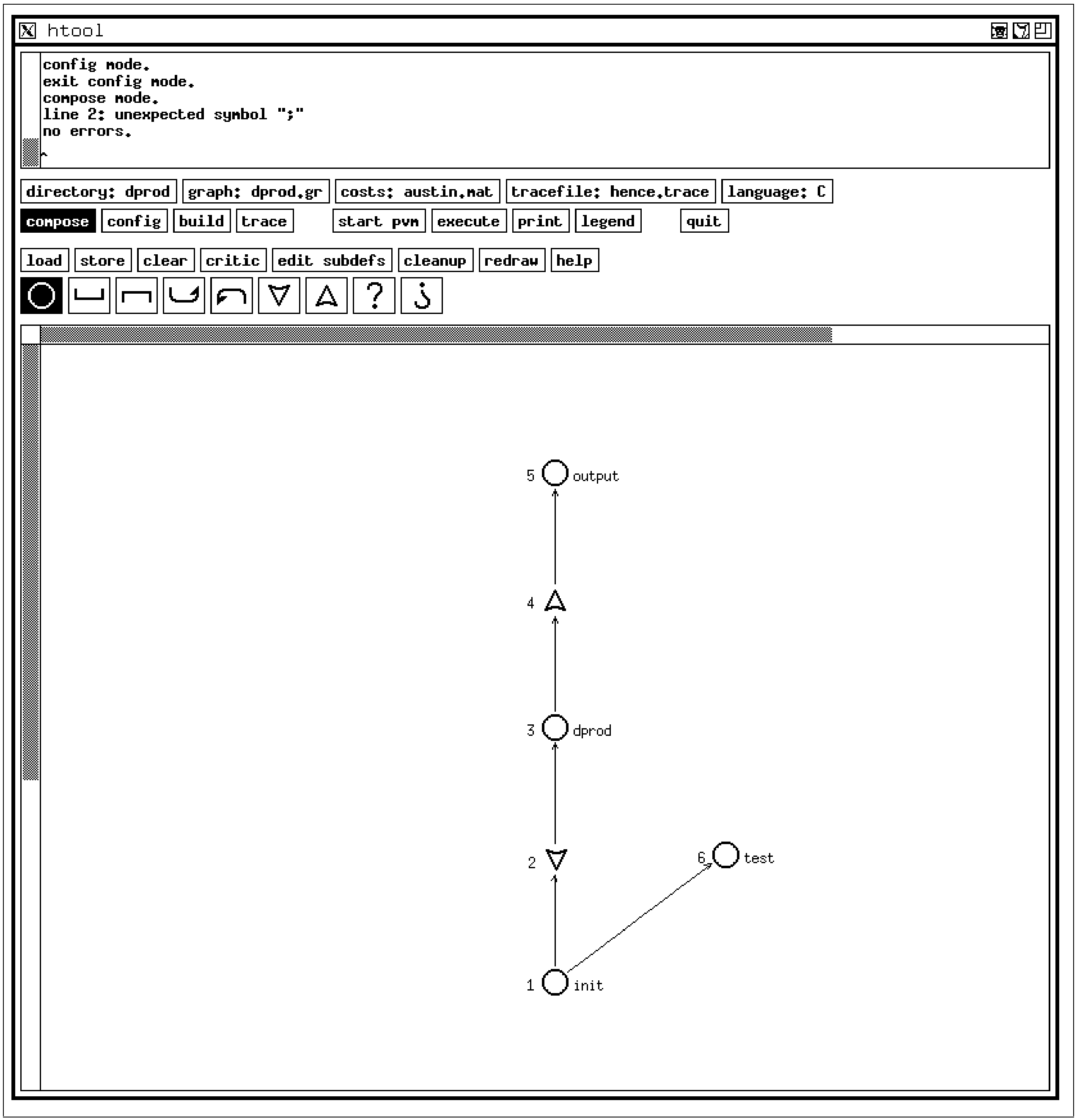

Figure 3: Compose mode.

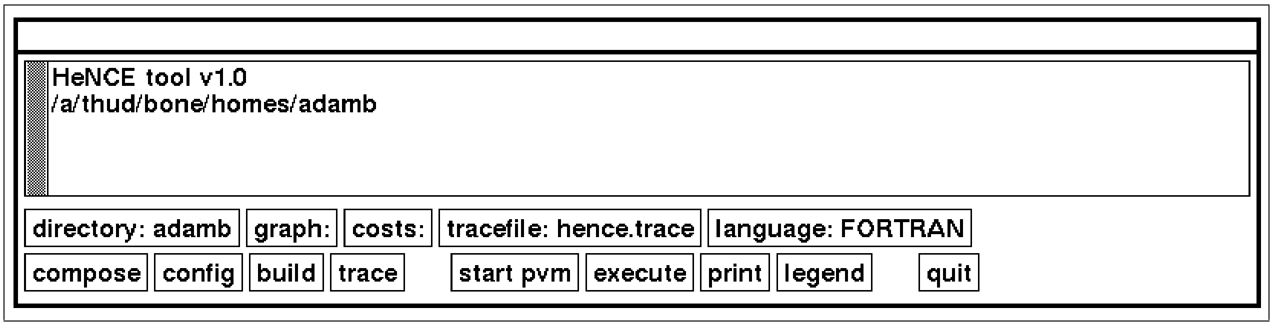

Figure 2: Main htool window.

There are two types of nodes in a HeNCE graph: normal nodes and special nodes. Normal nodes are used to indicate computation, and have node subroutines (implemented in C or FORTRAN) associated with them that perform the user's computation. Special nodes are used in pairs to delimit subgraphs of the HeNCE graph; they allow a portion of the graph to be repeated conditionally, sequentially, or in parallel during execution of the program. Most nodes in a HeNCE graph can have node programs attached. (Section 2.4 defines what the differences between node programs and node subroutines.) It is important to understand the difference between a node program and a node subroutine. The node subroutine is the subroutine which is invoked from the node program. A node subroutine may be called from multiple node programs.

The `load` and `store` buttons in compose mode allow you to load in and store HeNCE graphs. The `clear` button clears the current graph, removing all nodes and arcs. The `critic` button will parse the graph and report any irregularities that may exist (e.g. cycles in the graph). The `edit subdefs` button starts an edit session of the sub.defs file. The sub.defs file contains function prototypes for the node subroutines. These prototypes are described more detail later. The `cleanup` button will try to rearrange the graph based on a graph redrawing algorithm built into htool. You may want to store your graph before trying cleanup so you may revert to the original layout in the case that you do not like the automatically generated one. The `redraw` button will simply redraw the graph in the case it was not redrawn correctly in response to X expose events. Finally, the `help` button will display a window showing what mouse clicks to use when drawing a graph in compose mode. It is possible to change the actions associated with the various mouse clicks via X resources; however, the help display will not reflect the altered settings.

3.1.1 Drawing Graphs

Figure 3 shows htool in compose mode. The lower row of symbols show the various nodes which may be used in drawing a HeNCE graph. The default mouse actions for drawing in compose mode are shown in Table 2. To add a node to the graph, select the type of node you wish to add and click the left mouse button in the compose window. To delete a node hold the control key and click the left mouse button on or near the node to delete. To move the node hold the shift key down and use the left mouse button to click and drag the node to its new location.

To draw arcs between nodes, click the middle button on the node at the tail of the arc, drag the mouse to the node at the head of the arc and release the middle button. To delete arcs, hold the control key and click the middle button on or near the arc. An arc may be moved by holding the shift key while clicking the middle button near one end of the arc, dragging it to another node, and releasing the middle button.

To edit the node program for a particular node, click the right button near a node. An edit window will appear containing the node program. The node program is parsed immediately after the edit is complete. If there are problems with the node program, error messages will appear in the message center window and htool will ask you if you would like to edit the node program again. (For a description of the syntax for

- Create a program graph †
  - Attach subroutines and parameters to nodes †
  - Write sequential node subroutine code §
- Configure a virtual machine
  - Input machine names †
  - Input estimated cost for subroutine/machine pairs †
- Build executables
  - Create a subdefs file describing the subroutine parameters †
  - Generate wrapper code ‡
  - Generate makefile ‡
  - Execute make on required machines †§
- Execute program †
- Trace program execution †

(†) done through htool. (‡) done by htool. (§) done by user.

Figure 1: Steps in writing a HeNCE program.

and the main htool window will appear on your screen. Figure 2 shows the main htool window. The top part the htool window is a scrollable text window called the InfoBox. This window displays general information and error messages for the user.

Below the message center are two rows of buttons. The first row shows the status of the current directory, graph, cost matrix, tracefile, and language settings. The current directory can be changed by clicking the `directory:` button and selecting a new directory. A new HeNCE graph can be loaded by clicking the `graph:` button and selecting the name of the graph file to load. The same is true for the cost matrix. Clicking on the `costs:` button and selecting the name of a cost matrix file will load a new cost matrix. By convention graph files have the extension `gr` and cost matrix files `mat`. The `tracefile:` button allows the selection of a file containing postmortem trace information. The last button is a toggle indicating the language HeNCE should expect for the user defined subroutines. Currently Fortran and C are supported.

The second row of buttons represent different HeNCE modes. Clicking the `compose` button places htool in compose mode, allowing the user to create or edit a HeNCE graph. The `config` button enters the configure mode where virtual machines can be configured. Build mode supports the creation of executables. Trace will visually animate the execution of a HeNCE graph. The start pvm button will create a parallel virtual machine based on the machine names set in config mode. Execute runs the graph on the currently configured virtual machine. Print will print a postscript version of the graph. The `legend` button displays a legend showing the meanings of different node colorings. We will now describe each of these functions in more detail.

3.1 Composing a Program Graph

Compose mode supports the creation and editing of a HeNCE graph, Figure 3. A complete HeNCE program consists of a graph, node programs for nodes in the graph, and node subroutines. Graphs describe the control flow and parallelism of a program. Graphs are defined by drawing them with the mouse.

entries. These entries are used by htool to tell the local pvmd how to start up pvmd processes on remote machines. The format of this file is as follows:

```
hostname [options]
```

where `hostname` is the name of the remote host as used in htool. Options can be used to change (a) how the remote pvmd is started (whether by the `rsh` command or, if the `pw` option is specified, by the `rexec()` library routine), (b) what user name to use on the remote machine (if different from the user on the local machine), (c) the location of the remote pvmd program (if not `/tmp/pvm/pvmd`). If no pvmHostFile is specified, or if a host specified in htool's cost matrix does not appear in the pvmHostFile, then htool will tell the local pvmd to use its default settings when starting up remote pvmd processes. For a compete description of the PVM hostfile options see the PVM documentation. Using the X resource `pvmd` you can customize which executable htool will use when starting the local PVM daemon. The default `pvmd` is `/tmp/pvm/pvmd`.

The X resource `subDefsFile` specifies the name of the sub.defs file to be used by htool. This file should contain the prototypes for all HeNCE subroutines to be installed in `$(HOME)/pvm/$(ARCH)`, so that HeNCE programs can link to them. The subDefsFile is used by the `critic` function in compose mode, and by the `make wrappers` function in build mode, to check that the actual parameters passed to subroutines called from the HeNCE graph match the formal parameters used in the subroutine.

When composing a HeNCE program, htool will start various editing sessions to allow you to edit node subroutines and node programs. The editor that htool uses for these editing sessions is set by the X resource `editorCommand` and defaults to `xterm -title %s -e vi %s`, indicating that an xterm is started containing a vi edit session.

The iconList resource specifies the name of a file which contains a list of hosts and the names of files containing the icons that are used to represent those hosts in the htool trace hostmap panel. By setting this resource you can control which icons are used to represent which machines during a program trace.

The resources exitColor, notReadyColor, startedColor, executingColor, completedColor, warningColor, and errorColor are used to specify colors of the ndoes when drawing graphs in compose and trace mode.

There are also X resources which control the user interface aspects of htool. Colors and fonts for various parts of htool may be specified.

The degree to which htool can be customized may seem overwhelming at first. One should remember, however, that all of the X resources have reasonable default values and if your installation of PVM and HeNCE is canonical then no changes to these resources should be necessary.

3 Using HeNCE

In order to use HeNCE, the programmer must go through four main steps:

1. Create a HeNCE graph describing the parallel computation,
2. Configure a parallel virtual machine to execute the program,
3. Build executables for the various node subroutines on the appropriate machines, and
4. Execute the program on the virtual machine. Optionally the programmer may also view an animated trace of the program while it is executing or after it has completed.

These steps are further outlined in Figure 1.

In order to get HeNCE going you need to execute htool. Simply type

```
% htool
```

Resource	Example value	Description
font	10x20	General htool font
infoBox.font:	10x20	InfoBox font
configMat.font:	10x20	Config matrix font
iconList:	~/htool_icons	Where to find machine/icon mappings
includeDir:	$(HOME)/hence/include	Include file directory for generated makefiles
pvmLib:	$(HOME)/pvm/$(ARCH)/libpvm.a	Location of PVM library
pvmd:	/tmp/pvm/pvmd	PVM daemon to use
pvmHostFile:	/dev/null	File of PVM options for various machines
binDir:	$(HOME)/pvm/$(ARCH)	Where to place compiled hence programs
henceLibDir:	$(HOME)/hence/$(ARCH)	Location of HeNCE libraries
language:	C	Default language setting for node subroutines
subDefsFile:	sub.defs	specifies the name of the subdefs file
traceFile:	hence.trace	specifies the name of the default trace file
xtermCommand:	xterm	location of the "xterm" program on local machine
exitColor:	maroon	node color for trace mode.
notReadyColor;	grey	node color for trace mode.
startedColor:	tan	node color for trace mode.
executingColor:	green	node color for trace mode.
completedColor:	magenta	node color for trace mode.
warningColor:	orange	node color for compose mode.
errorColor:	red	node color for compose mode.
verifyPop*background:	red	Color for verify popup window
fileverPop*background:	yellow	Color for file verify popup window
infoBox.background:	yellow	Background color for the infobox
configMat.background:	blue	Background color for the configure matrix
editorCommand:	xterm -title %s -e vi %s	Command to execute for editing

Table 1: Htool X resources.

HeNCE has also been successfully compiled and minimally tested on Sequent Symmetry (using a remote X server), NeXT, Cray, and Hewlett-Packard machines, and will probably build and run on these machines without significant modifications.

For the most up to date information on compiling and installing the HeNCE software, see the files `Compiling` and `READ-ME` in the the source distribution.

2.4 Definitions

In order to keep the terminology straight, let us first define the terms we will be using. *HeNCE* refers to the entire programming environment and paradigm. The *htool* program is the graphical interface which realizes the HeNCE programming environment. A *HeNCE graph* consists of the graph (nodes and arcs) that specify control flow and parallelism of a HeNCE program. A *node program* is the text that is attached to a HeNCE graph node. This text usually declares variables and calls a subroutine. A *node subroutine* is a function or subroutine, written in either Fortran or C, that is attached to a HeNCE graph node via a node program. Thus, a complete *HeNCE program* consists of the HeNCE graph, node programs, and node subroutines.

2.5 Setup and Customization of htool

In creating a HeNCE program we will need to run htool and its associated programs. The executable files htool, mkwrap, and master need to be in your path on the machine where you will be writing your HeNCE program. The executables htool, mkwrap, and master do not need to be on each machine where your HeNCE program will execute, only on the machine you are creating your HeNCE program.

Since htool is an X window based program its behavior can be customized by setting X resources. X resources are set by placing a line in your .Xdefaults file and typing the command `xrdb -load .Xdefaults`. For instance, htool has a language switch which indicates which language you will be using for your node subroutines, either Fortran or C. The default for this switch can be set to Fortran by placing the line:

```
htool.language: Fortran
```

in your .Xdefaults file. Many of the htool resources and their defaults can be found in Table 1.

HeNCE has a set of libraries that it uses. Each node subroutine will need to be linked with the HeNCE libraries. These libraries should be compiled for each architecture you plan on using when running your HeNCE program. Typically these libraries will be stored in some place like $(HOME)/hence/*ARCH*, where *ARCH* denotes the architecture of the machine, (i.e. SUN4, PMAX, CRAY) as defined by PVM. Htool needs to know where these libraries have been placed. You should make sure your X resource `henceLibDir` points to the directory where the HeNCE libraries have been placed. The default setting for `henceLibDir` is `$(HOME)/hence/$(ARCH)`. Since HeNCE programs ultimately access PVM, they must also be linked with a PVM library. The location of the PVM library to use can be set by the X resource `pvmLib` which defaults to `$(HOME)/pvm/$(ARCH)/libpvm.a`. There are also HeNCE related include files of whose location htool must be aware. You should make sure the X resources `includeDir` is set correctly. The default is `$(HOME)/hence/include`. Once a node subroutine has been compiled, it will be placed in a binary directory. This directory is indicated by the `binDir` X resource. The default is `$(HOME)/pvm/$(ARCH)` matches the location where PVM expects to find its node executables and should not be changed unless PVM is changed also. Htool uses this information when creating makefiles to compile and link node subroutines.

Before executing a HeNCE program, htool will start a virtual machine. This is done by creating a PVM host file and starting a PVM daemon. The PVM host file is created using defaults from a master PVM hosts file. For instance, you may have a different login name on your Cray than you do on your workstation. This can be set in the master PVM hosts file so that whenever htool starts a virtual machine including the Cray it will use the correct login name. The location of this master PVM host file is set via the X resource `pvmHostFile` and the default is `/dev/null` so if the user leaves it unspecified, htool will not complain but simply build a vanilla PVM hosts file. If specified, this names a file containing a list of PVM-style host

Once the imake file is customized for your particular machine, the following steps will install the HeNCE software.

1. Make sure the directory containing the X11 "binary" programs (/usr/bin/X11 on many systems) is included in your PATH environment variable.
2. To build the top-level Makefile, type `xmkmf`. If the xmkmf program does not exist on your system, type: `imake -DUseInstalled -Iconfigdir` where `configdir` is the directory containing the X11 configuration files for your system (e.g. /usr/lib/X11/config).

   (On a CRAY, if you are running SCC as your default C compiler, you need to set the environment variable IMAKECPP to point to the old-style C preprocessor. Normally this is `/lib/pcpp`. Do this *before* running xmkmf.)
3. Having built the top-level Makefile, now build all of the Makefiles for sub directories by typing `make Makefiles`. (Note the s at the end of `Makefiles`.)
4. If you have old object (.o) files left over from a previous HeNCE compilation on a different system, type `make clean` to remove them.
5. Now simply type `make` which should compile everything.
6. Typing `make install` should install things.

If you don't have X windows installed, and only need to compile the HeNCE object libraries so you can link HeNCE node programs, (i.e. you are running htool on another machine and only want to run node programs on this one), follow the following steps:

1. cd master
2. edit the Makefile.libs file to set the ARCH and HENCELIBDIR variables as appropriate for your system
3. type "make -f Makefile.libs clean", if necessary, to remove any old object or executable files that may emain from a previous compilation
4. type "make -f Makefile.libs"
5. to install, type "make -f Makefile.libs install"

HeNCE has been successfully built on the following systems:

- DECStation running Ultrix 4.1, 4.2, and 4.2A, and DecWindows. (For htool to compile with the vendor-supplied X Windows libraries, you *must* have the "Unsupported X11 Components" software subset loaded.) Htool should also build and run with MIT's X11R4 or X11R5, but this configuration has not been tested.
- IBM RS/6000 running AIX 3.1 (RIOS) and X11R5. (Some effort has been made to this to work with X11R4, but R5 is preferred. HeNCE will not build using the X11R3-based libraries supplied with AIX version 3.1. The libraries to be supplied with AIX version 3.2 may work.)
- Sun 3 running SunOS 4.1 (SUN3) and X11R4.
- Sun 4 running SunOS 4.1 or later (SUN4) and X11R4 or X11R5. Note: htool should be built with X11R5 libraries that have the patches in MIT fix-09 applied. On SunOS 4.1.1 systems, if the Sun "ld patch" has been applied, or if SunOS 4.1.2 is being used (which incorporates the "ld patch") you need to edit your X11 config site.def file before building the X11R5 libraries. See the instructions with MIT fix-09 for details.

a network of heterogeneous machines. The programmer does not need to use any parallel programming primitives and does not need to write any code beyond that supplied for the node subroutines.

HeNCE relies on the PVM system [2] for process initialization and communication. Programmers wishing to write explicit message passing parallel programs on a network of machines should explore using the PVM system directly. (Section 2.2 explains how to obtain the PVM software.)

HeNCE is intended for the programmer or scientist who wishes to exploit a collection of machines to which he has access, yet does not want to delve into the details of parallel programming. HeNCE allows the programmer unfamiliar with parallelism to glue together sequential subroutines with a picture, creating a parallel program. For a more general description of HeNCE see [1].

2 Installing and Running HeNCE

The goal of our research is to develop parallel programming tools that assist scientists, researchers and students in utilizing parallel computers. We encourage people to try HeNCE and to give us feedback. If you have suggestions or problems regarding HeNCE you may send email to hence@msr.ornl.gov.

2.1 Acquiring HeNCE Software

The HeNCE software is freely available from netlib. Netlib is an automated software server. Requests for the HeNCE software from netlib are made by sending an email message to netlib@ornl.gov containing the message “send index from hence”. An index of the available HeNCE software, User’s Guide, and how to obtain them will be sent back via return mail. The HeNCE source is available in either shar or tar format. Instructions on how to unpack the software from the mail messages are sent along with the software.

2.2 System Requirements

HeNCE runs under most UNIX systems. Htool, the HeNCE window interface, runs under the X Window System, version 11, from MIT. You must have either release 4 or release 5 of X Windows (including the Athena widget libraries as supplied by MIT) to compile and link the htool program. A compiled htool program may be used with an X display server (either a workstation or an X terminal) that supports version 11, release 3 or later functionality. It is possible to use HeNCE without htool but the visualization aspects of htool will be lost. This paper focuses on the use of htool. If you wish to use HeNCE without the visual interface refer to Section 4.5.

In order to execute HeNCE programs you must have installed the PVM software on the computers you intend to use. PVM is also free and available from netlib. Send email to netlib@ornl.gov with the message “send index from pvm” to request the PVM software if you haven’t already installed it.

2.3 Compiling the HeNCE System

HeNCE uses the X windows “imake” facility to build Makefiles that are correct for your particular system. Imake reads the file named Imakefile, along with config files that should already be installed on your system as part of X windows, and produces a Makefile that should be correct for your system’s configuration.

Before compiling HeNCE for the first time, you must edit the file imake.defs to change the definition of LIBPVM to point to the name of the library archive containing PVM executables. Also change the definitions of HENCELIBDIR to indicate where you want HeNCE libraries installed, and BINDIR to indicate where you want HeNCE executables installed. (The latter two are only used by “make install”; you can move the libraries and executables to other locations later if you wish without needing to recompile HeNCE).

If you are building HeNCE for a new system (one which HeNCE does not already support), you will also need to edit the file imake.defs to add code to set the ARCH variable, before compiling HeNCE. The ARCH variable should match the one used by PVM.

HeNCE: A Users' Guide
Version 1.2

Adam Beguelin
Jack Dongarra
Oak Ridge National Laboratory
and
University of Tennessee

G. A. Geist
Oak Ridge National Laboratory

Robert Manchek
Keith Moore
Reed Wade
University of Tennessee

Jim Plank
Princeton University

Vaidy Sunderam
Emory University

hence@msr.epm.ornl.gov

February 10, 1992

Abstract

HeNCE, Heterogeneous Network Computing Environment, is a graphical parallel programming environment. HeNCE provides an easy to use interface for creating, compiling, executing, and debugging parallel programs. HeNCE programs can be run on a single Unix workstation or over a network of heterogeneous machines, possibly including supercomputers. This report describes the installation and use of the HeNCE software.

1 Introduction

HeNCE is a tool that greatly simplifies the writing of parallel programs. In HeNCE, the programmer explicitly specifies parallelism between subroutines by drawing a graph where nodes in the graph are subroutines written in either Fortran or C. The programmer must supply the HeNCE graph and the source code for the subroutine nodes. HeNCE will automatically execute the subroutines in parallel (whenever possible) across