HeNCE: A Users' Guide
Version 2.0

Adam Beguelin
Carnegie Mellon University

Jack Dongarra
Oak Ridge National Laboratory
and
University of Tennessee

G. A. Geist
Oak Ridge National Laboratory

Robert Manchek
Keith Moore
Peter Newton
University of Tennessee

Vaidy Sunderam
Emory University

hence@cs.utk.edu

June 15, 1994

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 which run under the PVM system which is also described in this document. HeNCE programs can be run on a heterogeneous network of UNIX workstations. This report describes the installation and use of the HeNCE software.

1 About this Document

This document describes the HeNCE graphical parallel programming environment as well as its installation and use. It is organized as follows.

- Section 2 provides a brief overview of HeNCE, how is is used, and who may be interested in using it.
- Section 3 lists the hardware requirements for running HeNCE.
- Section 4 describes how the latest version of HeNCE may be obtained and how its authors can be contacted for support.
- Section 5 describes the default HeNCE installation proceedure.

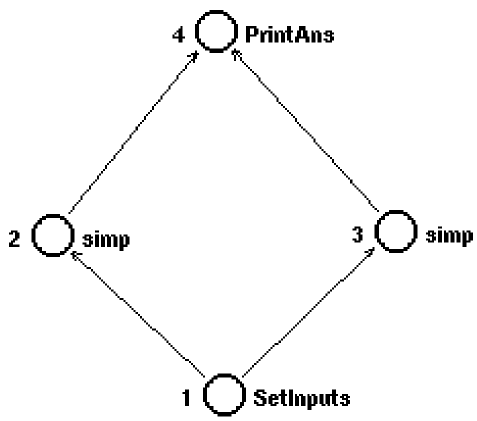


Figure 1: Example HeNCE Program

- Section 6 discusses how to create HeNCE programs and describes the syxtax of all node attributes.
- Section 7 defines cost matrices and describes how to create and use them.
- Section 8 describes how to build and run a HeNCE program once it has been entered.
- Section 9 discusses the HeNCE program execution trace facility.
- Section 10 provides a detailed tutorial example of using HeNCE.
- Section 11 describes how htool's appearance and operation can be customized.
- Section 12 provides some hints on how to use HeNCE in non-standard PVM environments.
- Section 13 describes how to compile the components of HeNCE from source distributions.
- Section 14 contains some trouble shooting hints.
- Section 15 lists known bugs and limitations.

2 Introduction

HeNCE is a tool that greatly simplifies the writing of parallel programs. In HeNCE, the user creates a parallel program by drawing and then annotating a picture that explicitly specifies a parallel computation. There are no traditional parallel programming primitives to learn.

The program is then automatically translated into an executable parallel program that runs under the PVM system as will be described below. The parallel program executes on a network of one or more UNIX workstations. The workstations need not all be of the same type.

The picture the user draws is a directed graph in which nodes represent calls to conventional sequential subroutines written in C or Fortran. Arcs represent execution ordering constraints. A node may execute its subroutine when all nodes that preceed it in the graph have executed theirs. Figure 1 shows a simple example HeNCE program that uses only a small subset of HeNCE's facilities.

The program begins by executing (or firing) node 1. Notice that HeNCE graphs are, by convention, drawn from bottom to top. After node 1 fires, nodes 2 and 3 may execute in parallel. Finally, node 4 runs, and the program terminates. The name of the subroutine that a node runs is show next to it. For example, nodes 2 and 3 both run separate invocations of subroutine `simp`.

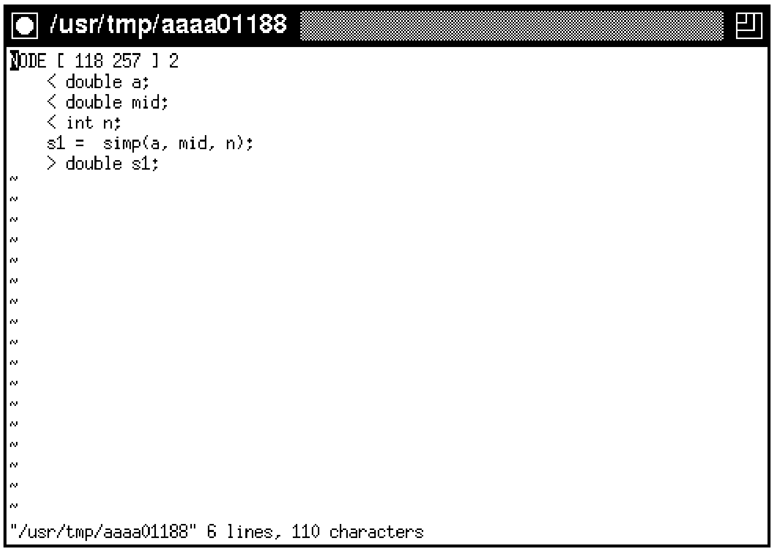


Figure 2: Annotation for Node 2

The steps a user must perform to create, compile, and run a HeNCE program are as follows. These steps, as well as the other facilities of the HeNCE language, will be described in detail later.

1. Draw a picture that shows the desired parallel structure. The drawing environment is called htool. It is mouse-based and runs on workstations that support X windows (X11R4 or later).
2. Annotate the nodes in the picture. Annotations are entered into a pop-up text editor window. Figure 2 shows the annotation for node 2 of Figure 1. The annotation shows the call to the sequential subroutine as well as what variables from predecessor nodes are read and what variables are made available to successor nodes. In Figure 2, variables `a`, `mid`, and `n` are set by node 1, and variable `s1` is read by node 4.
3. Write all of the sequential subroutines that will be called from the HeNCE nodes.
4. Define the virtual parallel machine on which the parallel program will run. This is done by specifying a cost matrix that lists the cost of running each of the program's subroutines on each of the hosts the user selects for use by the parallel computation. The cost matrix thus defines both the list of hosts on the network that make up the virtual parallel machine and provides HeNCE with execution cost estimates for each subroutine that help it to run each subroutine on a host that is chosen to maximize overall performance. The default cost matrix runs all nodes on the machine on which you are running htool.
5. Select `Build` from the `Build` menu to compile the parallel program.
6. Select `Run Program` from the `Run` menu to execute the parallel program.
7. HeNCE will animate the program's execution and will produce a strip chart that shows the utilization of the hosts in the virtual parallel machine during the run.

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 also [1]. HeNCE is non-exclusively oriented towards applications in computational science.

3 Supported Hardware

There are two phases to HeNCE use. First a parallel program must be drawn and annotated. Then, it must be run on a virtual parallel machine. It is possible to use different machine types for these two phases.

The HeNCE program creation environment (htool) runs on UNIX workstations that support X Windows version X11R4 or X11R5 (including the Athena widget libraries). It has been tested on the following workstation types, and will likely work on others as well. Sun Solaris is not supported at this point, but support for it is planned in a future release. If we receive email requests for it, we will likely move faster.

```
Architectures that Support htool (to Create HeNCE Programs)
  Architecture      Operating System     PVM Arch Name

  Sun 4 (Sparc)        SunOS 4.1.3          SUN4
  IBM RS/6000          AIX 3.2.5            RS6K
  HP Precision Arch    HPUX                 HPPA
  And some others...
```

HeNCE programs run under PVM so HeNCE programs may be run on any heterogeneous network of machines on which PVM is supported.

```
Architectures that Support PVM Execution (to Run HeNCE Programs)
  Architecture      Operating System     PVM Arch Name

  Sun 4 (Sparc)        SunOS 4.1.X          SUN4
  Sun 4 (Sparc)        Solaris 2.2.X        SUN4SOL2
  Sun 3                SunOS 4.1.X          SUN3
  IBM RS/6000          AIX 3.2.X            RS6K
  DEC (Alpha)          OSF-1                ALPHA
  DEC (PMAX)           Ultrix               PMAX
  HP Precision Arch    HPUX                 HPPA
  And many others...
```

The current release of HeNCE is oriented towards running programs on a network of sequential machines. It does not make completely effective use of parallel machines such as the Connection Machine 5, Intel Paragon, and others, even though PVM supports these machines.

3.1 The Default HeNCE Environment

While it is possible to use HeNCE in other ways, it is simplest if you adopt the default HeNCE environment which is defined by the following criterea. It is the operating environment adopted by many institutions.

1. Your working environment consists of a network of computers running UNIX.
2. You can rlogin or rsh from one computer on your network to another without entering a password. If this is not the case, you may want to enter the command "man rhosts" and read the UNIX man page about the .rhosts file. Also, see the section on non-standard PVM environments below.
3. You will place all HeNCE and PVM software within a directory called `pvm3` that resides within your home directory. HeNCE will also store executable images and ".o" files for your parallel programs within this directory.

If you do not have a collection of machines that satisfies these criteria, you might want to note that an environment consisting of single workstation on which your home directory is mounted does. if you are new to HeNCE, you may want to consider starting by creating and running your HeNCE programs all on a single workstation. Of course, there will be no parallelism since all subroutines will run on the same computer, but you will be able to learn about HeNCE and decide whether it will be worth the extra trouble of configuring it for a non-default environment.

3.2 What is PVM?

This document is about HeNCE, but since HeNCE uses PVM as its execution environment, some explaination of it is in order. PVM [2] is a set of library routines and control programs that allow distributed programs to be created using either "C" or Fortran. Users add calls to PVM routines to otherwise sequential programs to perform tasks such as message passing and process creation.

It is possible to write distributed programs using PVM directly rather than using HeNCE. Indeed many people do since PVM is relatively easy to install and use in comparision to many other message passing distributed computing libraries. PVM documentation is available from the same sources as the PVM software listed above.

4 Obtaining HeNCE and Support for It

This section describes the terms under which HeNCE is made available, lists the packages you will need to use HeNCE, and describes how the software may be acquired.

4.1 Terms of HeNCE Distribution

HeNCE and the PVM package that it requires are distributed free of charge electronically via the Internet, as described below. HeNCE is a research project. We cannot guarantee its fitness for any particular task, nor can we guarantee ongoing support, although we intend to try to provide it.

4.2 HeNCE Support and Questions

HeNCE and PVM are the joint work of many researchers at many sites. Questions and comments about the system and its use with PVM may be sent via internet email to `hence@cs.utk.edu`. If you try HeNCE, please send us a note describing your use of it, whether your experience is positive or negative. Your feedback is appreciated and is necessary to our research.

If you do encounter a problem please include the machine type and operating system release level for both the machine on which you run htool and all machines in your virtual parallel machine.

There is also a USENET newsgroup `comp.parallel.pvm` which is devoted to discussion of all aspects of PVM use, including HeNCE.

4.3 Packages Required to Use HeNCE

In order to use HeNCE, you must acquire and install several different components of the HeNCE environment. You need PVM version 3.2.X, The HeNCE programming environment called htool, and the HeNCE runtime support libraries. We offer this software packaged in a number of different ways.

1. PVM 3.2.4 or later.

   PVM is available in source form only. The current release of PVM is 3.3.

2. hence-2.0-src.tar.Z.uu

   This is the complete source code for htool, its runtime support libraries, and examples. If you wish to compile HeNCE, this and PVM are all you need. We suggest, however, that you obtain a pre-compiled version of HeNCE to save you the trouble of building the software.

3. hence-2.0-full.ARCH.tar.Z.uu

   ARCH is the name of an architecture such as SUN4 (Sun Sparc running SunOS) or RS6K (IBM RS/6000). These packages contain htool and the HeNCE runtime libraries precompiled for a particular architecture. Suppose you wish to use IBM RS/6000s to write and run HeNCE programs. Then, all you need is hence-2.0-bin.rs6k.tar.Z.uu and PVM.

   If you have trouble running pre-compiled HeNCE, we would very much like to receive email from you describing the problem. Please send email to hence@cs.utk.edu and provide the version of HeNCE you have, the version of PVM you have, what kind of machine(s) you are using and what release level their operating systems are. If you do not know all of this, send us email anyway.

If you cannot find a pre-compiled version of HeNCE for the machine that you wish to use, please send us email, and we will provide it if we can. We encourage such requests since they help us to know which machine types we should support.

4. hence-2.0-rtl.ARCH.tar.Z.uu

   Again, ARCH is the name of an architecture. These packages contain only the runtime libaries pre-compiled for a specific architecture. For example, if you will use SUN4 machines as part of your virtual parallel machine, but will never run htool directly on a SUN4, you could get hence-2.0-rtl.sun4.tar.Z.uu rather than hence-2.0-full.sun4.tar.Z.uu in order to save disk space.

5. hence-2.0-examples.tar.Z.uu.

   This package contains several example HeNCE programs that you can build and run. It is not required that you download it.

6. hence-2.0-doc.ps.Z.uu

   This package contains HeNCE documentation in Postscript form.

4.4 Methods for Obtaining HeNCE Software

There are three different ways of obtaining the necessary packages from our electronic distribution machines. We recommend against obtaining HeNCE and PVM (especially in binary form) from any sites other than ours.

1. Xnetlib. If you have xnetlib [3] installed, you can obtain the HeNCE packages using it. Xnetlib is a graphical interface to a large library of software and documents that are of interest to computational scientists. Xnetlib itself can be obtained by means of netlib or by anonymous ftp to ftp.netlib.org. PVM is in the pvm3 directory, and hence files are in the hence directory.

2. Netlib. Netlib is an email server to the same repository used by xnetlib. Using it, you can obtain all HeNCE packages via email. 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. Since the files you will get are large, email is an inconvenient way to receive them.

3. Anonymous ftp. All HeNCE packages are available by anonymous ftp to ftp.netlib.org [128.169.92.17]. PVM is in the pvm3 directory, and hence files are in the hence directory.

5 Installing and Testing HeNCE

This section describes how to install and test HeNCE. We assume that you will have obtained HeNCE pre-compiled. Instructions for compiling it from source can be found at the end of this document.

1. PVM must be compiled and installed before HeNCE. Detailed instructions come with PVM, but the process is quite simple. Unpack the source file in your UNIX home directory. Software will be placed into directory pvm3. CD to directory pvm3 and enter "make" to build and install PVM. HeNCE requires that PVM be installed within directory pvm3 within your home directory. Do this first!

   If you are using PVM 3.3 or later you muat ensure that environment variable PVM_ROOT is set to "~/pvm3" automatically whenever you login. For example, csh users will need to add

   ```
   setenv PVM_ROOT ~/pvm3
   ```

   to their .cshrc file.

2. HeNCE pre-compiled distribution files should be unpacked in your UNIX home directory. Software will be written into directory pvm3.
3. Be sure that "~/pvm3/bin/ARCH" is in your UNIX path for each machine that will be used with HeNCE in any way. "ARCH" is an architecture name.
4. Ensure that you can login without entering a password to all of the machines that will be a part of your virtual parallel machine from the machine on which you will run htool. See the "Default HeNCE Environment" above.

If you are confused about how to unpack HeNCE or PVM files, you should probably seek the advice of a local UNIX guru, but here are some hints. Suppose the file you wish to unpack is called NAME.tar.Z.uu where "NAME" is some prefix. First enter "uudecode NAME.tar.Z.uu" to create file NAME.tar.Z. Then, enter

```
zcat NAME.tar.Z | tar xf -
```

to unpack the file.

5.1 Testing the HeNCE/PVM Installation

Assuming you obtained them, A good way to test that HeNCE and PVM are correctly installed is to build and run one of the example programs that is supplied with HeNCE.

1. Use the UNIX cd command to change to directory "hence-examples/static_integ" and enter "ls" to be sure that files `PrintAns.c`, `SetInputs.c`, `simp.c`, and `static_integ.gr` are present. The directory hence-examples is created when you extract the files in hence-2.0-examples.tar.Z.uu.
2. Run htool in the background by entering "htool &" and then pick "Load" from the File menu in the HeNCE Graph part of the htool window. Click on file static_integ.gr so that it appears in the filename box and then click on OK. A HeNCE graph should appear in the window. This program integrates 1/x from 1.0 to 10.0 in parallel by having one `simp` node integrate from 1.0 to 4.5 while the other integrates from 4.5 to 10.0. The results are then summed and printed by node `PrintAns`. This is a toy example to get you started.
3. Build the program by selecting "Build" from the Build menu on the PVM Console half of the htool window. Watch for error messages, but there should be none.
4. Run the program by picking "Run Program" from the Run menu on the PVM Console half of the htool window. The program will be animated as it runs. By default, the program will run on the machine on which you are running htool.
5. When the program finishes, it will print the answer in the PVM Console window. At this point, pick "Exit Run Mode" from the PVM Console Run Menu.
6. Pick "Halt PVM" from the PVM menu of the PVM Console half of the htool window. If you fail to do this, you will leave PVM daemon processes running when you exit htool.
7. Exit htool by picking "Quit" from the HeNCE Menu at the upper left of the htool window, just above the "mode:" text line.

At this point, you should probably read the rest of this manual before attempting to do much more with HeNCE. There is a detailed tutorial example below that guides you through creating a program and running it on multiple machines.

6 Creating HeNCE Programs

This section explains how to write HeNCE programs. It describes the operation of the user interface, and it defines all of HeNCE's node types and their annotations. It also discusses some aspects of array usage in HeNCE, both from C and from Fortran.

6.1 The HeNCE GUI

HeNCE programs are created within a graphical environment called htool. In order to run it, your workstation must be running X windows version X11R4 or later. Start the htool application by entering the command "htool." No command line arguments are necessary, but htool does recognize the standard X Windows arguments that control such things as window geometry.

When you run htool, a single large window will appear. As is shown in Figure 3, this window is divided into a number of segments, each of which has a special purpose.

To `quit htool`, click the left mouse button on the button that says "HeNCE tool v2.0" and is located at the upper left of the htool window. A menu with three options will appear.

1. Change Directory. Change the directory in which htool will look for and save files.
2. Change Language. Change the language that will be used to express the sequential subroutines that nodes will call. All nodes in a HeNCE program must call subroutines written in the same language.
3. Quit. Exit the htool environment. Before quiting, you must save any changes you make to graphs or cost matrices or they will be lost.

The HeNCE button is at the top of a window segment that displays status information such as the current UNIX directory, the current graph, cost matrix, and trace file names, and the currently selected language for the subroutine that nodes will call.

The segment titled "Information" is to the right of the status segment, at the top of the window. Error messages, warnings, and other useful information will appear here.

The lower portion of the htool window is divided in half. The left half is devoted to drawing and displaying the HeNCE graph. The right half is devoted to the PVM virtual parallel machine. The top segment within it contains the cost matrix. The bottom segment is a strip chart that will show host utilizations when the HeNCE program is run.

6.2 Drawing HeNCE Nodes

HeNCE graphs are drawn in the large initially empty segment in the lower left of the htool window. The mouse is used to draw nodes and arcs.

To draw a node, you must select the type of node you wish to draw by clicking the left mouse button on one of the icons to the left of the drawing area. The selected icon will become highlighted. Then, click the left mouse button on an empty area within the graph drawing segment of the window. A node will appear. The node type selection will remain active until it is changed by clicking on a different icon. The meaning of the different icons will be discussed below.

The meaning of clicking a mouse button is obvious. Press the button and immediately release it. To "drag" with a mouse button, move the mouse cursor over an object, then press a mouse button and hold it down. Move the mouse, while continuing to hold down the mouse button. and then complete the drag by releasing the mouse button.

To draw an arc from one node to another, drag from one node to another with the middle mouse button. The arc will appear. There is no way to create intermediate points on arcs; the arc must go straight from one node to another.

Other graph editing operations involve holding down a modifier while clicking or dragging with the mouse. For example, to move a node, hold the shift key down and drag the node with the left mouse button. To move an arc, hold the shift key down and drag the arc with the middle mouse button. Table 1 summarizes.

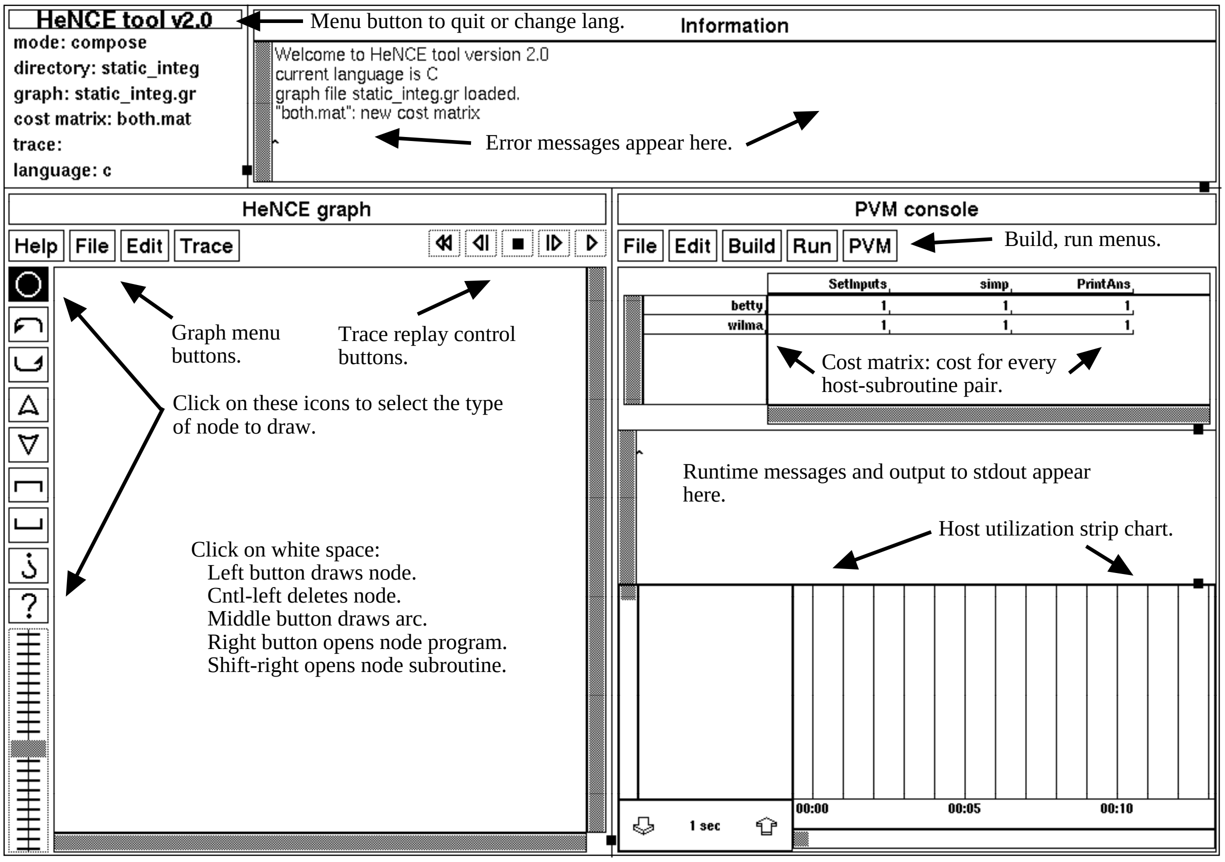


Figure 3: The htool Window

	Left	Middle	Right
None	Add Node	Draw Arc	Edit Node Program
Control	Delete Node	Delete Arc	undefined
Shift	Move Node	Move Arc	Edit Node Subroutine

Table 1: Graph Editing Operations

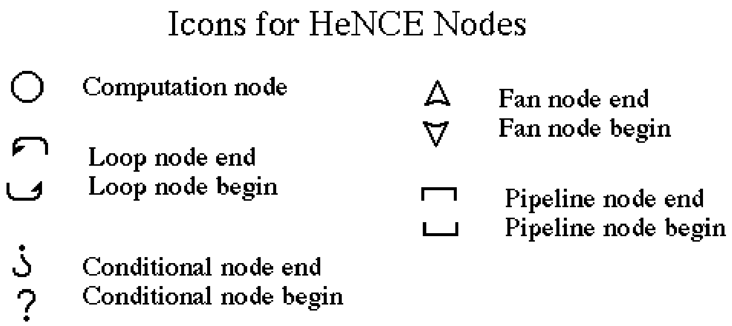


Figure 4: The htool Window

6.3 Definitions

A “node program” is the annotation you enter for a node. For computation nodes, it consists of a description of what variables the node uses and what subroutine it calls. A computation node's “node subroutine” is the text of the subroutine it calls.

If you click the right mouse button on a node, a text editor window that contains its node program will appear. If you shift-click the right mouse button on a computation node, a text editor window that contains the node subroutine will appear. Simply exit the editor normally (ZZ for vi) to make these windows go away. See the section on customizing HeNCE if you wish to use an editor other than “vi”, the default.

6.4 HeNCE Node Types

This section provides an overview of the different types of nodes that may appear in a HeNCE program. We will first summarize their use and then delve into the details of their node programs.

6.4.1 Overview of HeNCE Node Types

The HeNCE language contains a number of different node types. Each type is represented by an icon. Figure 4 shows all of HeNCE's node types. Notice that all but one of the icons come in pairs.

The circle icon represents a sequential computation node. It calls a user-supplied subroutine when it executes. This node type represents the basic unit of parallelism in HeNCE. Sequential computation nodes may fire when all of their predecessors in the graph have fired.

The icons that come in pairs represent “special-purpose” nodes. These generally define control flow within the HeNCE graph. There must always be a subgraph between a “begin” and an “end” node. The special-purpose nodes can be thought of as graph rewriting primitives. They add subgraphs to the current HeNCE graph at runtime, based upon expressions that are evaluated at runtime.

It is legal to include special-purpose nodes within the subgraph of some other special-purpose node. For example, one can place a fan inside a loop. However, the constructs must be nested. It is not legal to interleave them.

Conditional node pairs define an “if-then” structure. The Conditional-Begin node program contains an expression. If the expression evaluates to TRUE (meaning non-zero, following the C language convention), the subgraph between the pairs is executed. Otherwise, it is not. Note that HeNCE does not contain an “if-then-else” structure. In terms of graph rewriting, the subgraph is added to the graph if the expression is TRUE. Otherwise, it is replaced by an arc. See Figure 5.

Loop node pairs define iterations much like a C “for” loop. The subgraph between the loop begin and the loop end node is repeated as an index variable takes on successive values. Viewed in terms of graph rewriting, the subgraph is replaced by a sequence of copies of itself. See Figure 5.

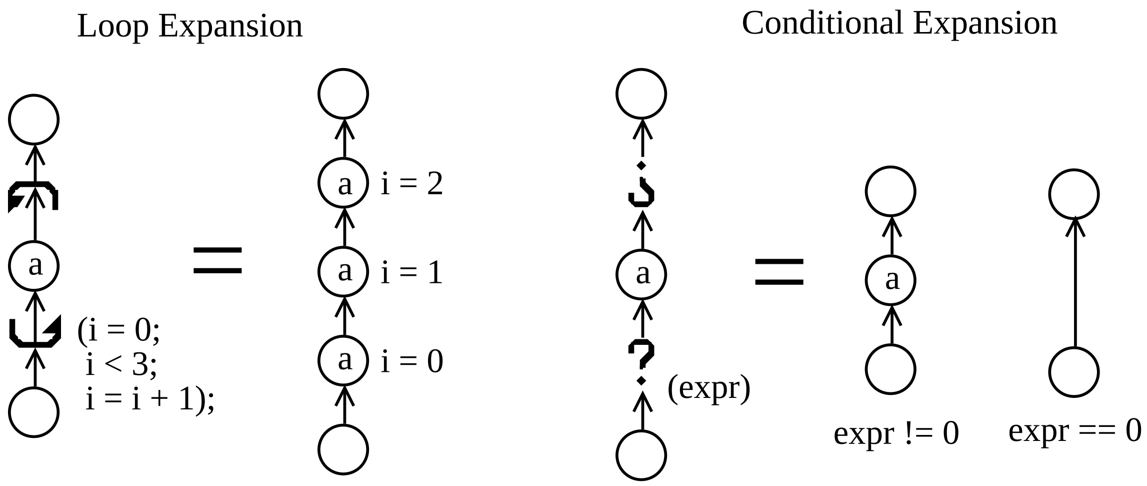


Figure 5: Effects of Conditional and Loop

Fan node pairs create parallel structures. They replicate the subgraph between them and evaluate the replications in parallel. Figure 6 shows the effect of a Fan. The number of nodes created is determined by the Fan Begin node's program and can depend on values of runtime variables.

Pipe node pairs create a pipeline structure. Figure 6 also shows the effect of a pipe and delineates which nodes within the pipe can execute in parallel. The subgraph within the pipe is replicated, somewhat like a Fan node, but the dependence structure differs.

6.4.2 HeNCE Node Programs

The precise action of a HeNCE node must be defined by the user. For example, a computation node executes a sequential subroutine, but the user must specify which subroutine. This is done by means of a textual description called the node program.

The syntax of node programs varies among HeNCE's node type. Computation nodes are the most complex and so will serve as the starting point. Discussion of special-purpose nodes will follow.

Computation Node Programs Computation node programs consist of three parts, but not all nodes will need all three. The purpose of the parts is described in the list below. The parts must be entered into nodes in this order.

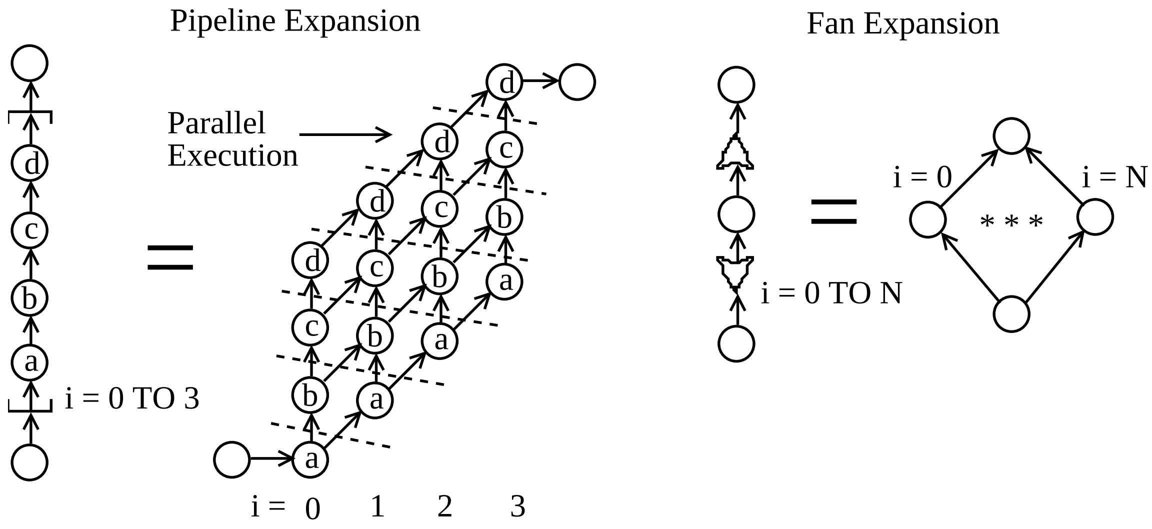
1. `Input Declarations.` The input declarations specify which variables the node will use. These variables are defined and set by predecessor nodes in the graph.

2. `Subroutine Call.` The subroutine call is what it sounds like– a call to a sequential subroutine that will be executed when the node fires. The call includes both the subroutine name and all actual parameters.

3. `Output Declarations.` The output declarations part defines new variables that are declared and set within the node and that will be made available to successor nodes in the graph.

Input declarations Input declarations are of the form:

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



:

Figure 6: Effect of Fan and Pipe

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 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:

```
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:

```
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:

```
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 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.

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. Currently, HeNCE stores arrays in row major. Thus multidimensional arrays do not work as expected in FORTRAN – the programmer is advised to stick to single-dimensional arrays.

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:

			subroutine	
		dprod	init	output
host name:	betty	4	2	6
	wilma	1	3	2

Figure 7: Example Cost Matrix

```
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.

7 Defining a Cost Matrix

A cost matrix does two things. It defines the list of host computers that make up the abstract parallel machine that will run the HeNCE program, and it provides HeNCE with an estimate of the cost of running each of the program's subroutines on each of the hosts. This helps the HeNCE system make intelligent choices about which host to use to run which subroutine.

7.1 Default Cost Matrix (Sequential Execution)

If you do not supply HeNCE with a cost matrix when you either build or run a program, it will use a default in which the all nodes will run on a single host– the host on which htool is running. Of course, this means that no parallelism will be realized since only one processor is available. This mode is convenient when first learning HeNCE since you can run programs while completely ignoring the issue of cost matrices. It also may be helpful during program debugging.

7.2 Entering a Cost Matrix for Parallel Execution

If you wish to exploit true parallel execution of your HeNCE program, you must allocate more than one host to your abstract parallel machine. To do this, you must define a cost matrix.

A cost matrix is a table, the rows of which are keyed by host names and the columns of which are keyed by subroutine names. There must be exactly one row in the cost matrix for each host in your parallel abstract machine, and there must be exactly one column for each subroutine called from your HeNCE graph. The enties in the cost matrix are integers that represent the relative cost of running a given subroutine on a given host.

Figure 7 shows an example cost matrix. Note that there are two hosts in the abstract parallel machine. Their names are betty and wilma. A HeNCE program that uses this cost matrix will run using only these two machines. The HeNCE program contains three sequential subroutines that are called from nodes: `dprod`, `init` , and `output`.

The cost matrix shows that it is three times as expensive to run subroutine `output` on betty as it is on wilma, but `init` is cheaper by a factor of two thirds on betty. HeNCE will use this information to try to run subroutines on the hosts on which they run cheapest.

The defintion of cost is arbitrary. HeNCE simply uses the numbers you provide. Most likely, you will choose to associate the costs with the subroutines' execution times on various machines. Unfortunately, HeNCE provides no support for choosing costs. It does not measure the execution costs. You must estimate them somehow.

If you are running your program on an network of machines with similar performance characteristics (perhaps because all machines are the same kind and have similar loads), it is perfectly reasonable to give all of the costs in the matrix the same value. HeNCE will try to distribute the nodes onto hosts in such a way that good parallelism is obtained.

Note that the hosts in the cost matrix can be of any machine type for which you have installed PVM and the HeNCE libraries. HeNCE will automatically compile the subroutines for the machines on which it might execute them.

7.3 Cost Matrix User Interface

This section describes the mechanics of entering a cost matrix. Cost matrices are entered at the top of the PVM console area of the htool window in the scrolling region reserved for them. The File and Edit menu buttons in this area operate on cost matrices.

The easiest way to enter a cost matrix is to leave it blank until your HeNCE graph is completely drawn and all node programs are entered. At this point, HeNCE will know the names of all subroutines that are called from computation nodes. From the PVM Console Edit menu, select "Add from Graph" and a row for your first host will be added to the previously empty cost matrix. Fill in the host name and all costs for this row. Then pick "Add from Graph" to create the row for the second host, and so on.

When the matrix has been entered, save it into a file by picking "Save" from the PVM Console file menu. Do not forget to do this before exiting htool. The cost matrix is not automatically saved. By convention cost matrix files end with ".mat", but this is not required.

To enter text into a field within a cost matrix, simply move the mouse to the field and type. The text will appear. Use the delete or backspace key to erase characters. Some other cost matrix manipulation functions follow.

1. To load a previously saved cost matrix, pick "Load" from the PVM Console File menu.

2. To create a blank new untitled and unsaved cost matrix, pick "New" from the PVM Console File menu. Your current cost matrix will be lost unless you save it first.

3. To add a row for a new host, pick "Add Host" from the PVM Console Edit menu.

4. To add a column for a new subroutine, pick "Add Subr" from the PVM Console Edit Menu.

8 Building and Running a HeNCE Program

Compiling a HeNCE program is very easy. Simply pick "Build" from the PVM Console Build menu. Your program and subroutines will automatically be compiled on all necessary machine types as defined by the current cost matrix. If there are errors, messages will appear in the Information segment of the htool window or perhaps in the center of the PVM Console segment.

If you change either your HeNCE program, a sequential subroutine, or the current cost matrix, you should rebuild before attempting to run your program again.

Running programs once they are compiled, is equally easy. Pick "Run Program" from the PVM Console Run menu. As the program runs, its progress will be animated. The icons that make up the HeNCE graph will change to show the node's state. Figure 8 defines the meening of the changed icons.

The state of abstract parallel machine is displayed in the lower right of the htool window during the program run. There is a strip for each host. The meaning of the strips is color coded as shown in Table 2.

Output the program writes to the UNIX standard output (stdout) will appear in the scrolling region of the PVM Console window segment.

Once your program begins running, it will remain in run mode until you pick "Exit Run Mode" from the PVM Console Run menu. Until you do this, you will not be able to edit your HeNCE graph, etc.

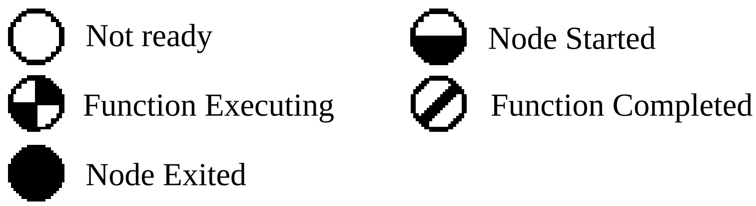


Figure 8: Meaning of Runtime and Trace Icons

Color Display	Mono Display	Meaning
Green	Dark Gray	Node Running
Yellow	Light Gray	No Node Running
Gray	White	No Task

Table 2: Abstract Parallel Machine State Display Colors

Trace information is written, by default, to a file called "hence.trace." This can be changed by picking "Set Trace File" from the PVM Console Run menu. Trace data is always saved. As is described below, it can be used to replay the program's animation.

In order to run your program, the necessary PVM demons must also be running. If they are not, HeNCE starts them for you automatically. Before exiting htool, you will probably want to halt them by picking "Halt PVM" from the PVM Console PVM menu. If you do not, you will likely leave unwanted processes running on all of the hosts in you cost matrix as well as the host that is running htool.

It is quite possible that your HeNCE program will not run to completion due to bugs. For example, one of your sequential subroutines might contain a bug that causes a divide by zero or a segmentation violation. If this occurs you may want to pick "Kill Program" from the PVM Console Run menu followed by "Exit Run Mode."

9 The HeNCE Trace Facility

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.

To view a program trace enter trace mode of htool by selecting the **Load Trace File** option under the **Trace** pull-down menu. The graph and cost matrix with which the program was run will be displayed.

The row of buttons just above the graph 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 reverse step button. It causes the graph to single-step backwards in time. The third button is the stop button. It can be used to stop playback of the tracefile. The fourth button is used for single stepping forward through the tracefile. The fifth button is the play button, used to continuously play back the tracefile.

To the lower left of the graph is a slider. This slider determines the speed of the trace playback. At the center, the trace will be played back in approximately real time. The top setting is 100 times faster, and the bottom setting is 100 times slower.

During the trace, nodes in the program graph and the machines in the cost matrix are highlighted to indicate their state. The histogram at the lower right of the window also displays the state of each machine over time. The scrollbars on the histogram can be used to display different parts of the histogram; vertical to see different machines, and horizontal to scroll through time. The up and down arrows in the histogram are buttons that change the time scale. During tracing and execution, a red hairline indicates the current time.

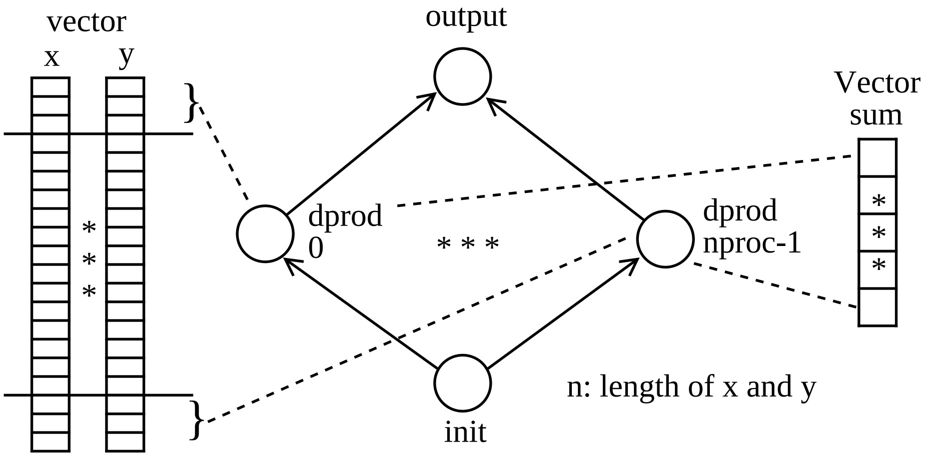


Figure 9: Parallel Dot Product Algorithm

10 A Tutorial Example

This section describes in great detail the process of creating, compiling, and running a simple HeNCE program that computes the dot product of two vectors in parallel. It is best if you actually run htool while reading this section and perform the steps as they are described.

The algorithm used in this example program is extremely simple. Two vectors are allocated. The dot product is computed by splitting the vectors into "nproc" subvectors. The dot products of the subvectors are computed in parallel and then the results are summed to form the dot product of the whole vectors. Figure 9 shows the parallel algorithm used. A HeNCE fan construct will be used to create the `dprod` nodes that run in parallel.

Node `init` allocates storage for `x` and `y` and sets `n` to their length. It also gives the arrays their initial values. Vector `sum` will be used to hold the results of the dot product computation on each of the subvectors.

Each replication of `dprod` will access a subvector of `x` and `y` and perform a dot product on them. The result will be stored in a single element of `sum`. Node `output` sums the entries in `sum` to form the final result.

Before beginning to create this HeNCE program, you should verify that you are running X windows (or perhaps Openwindows on a Sun) and that htool is in your command path.

1. The HeNCE examples package contains all sequential subprograms used in this example. This saves you the bother of typing them in. Go to UNIX directory "hence-examples/dprod" and enter "ls" to verify that files dprod.c, init.c, and output.c are there.

2. Begin htool by entering "htool &" at the UNIX prompt. This runs it in the background. An htool window with no graph should appear. If the current language is not set to "C", you must change it. The current language is shown in the text in the upper left of the htool window. You can toggle the current language between "C" and Fortran by picking "Change Language" from the HeNCE menu at the upper left of the htool window. This menu also allows you to quit htool.

3. The first step is to draw the HeNCE graph nodes. The eventual goal is to draw a graph that looks like the one shown in Figure 10. Do not worry if your node placement is not exact or if the node numbers on your screen do not correspond to those shown in the figure, although they will if you draw the nodes in order from bottom to top.

   First draw the nodes. With the left mouse button, click on the circle icon on the left of the screen. This ensures that Computation nodes are currently selected for drawing. (It should be the default.)

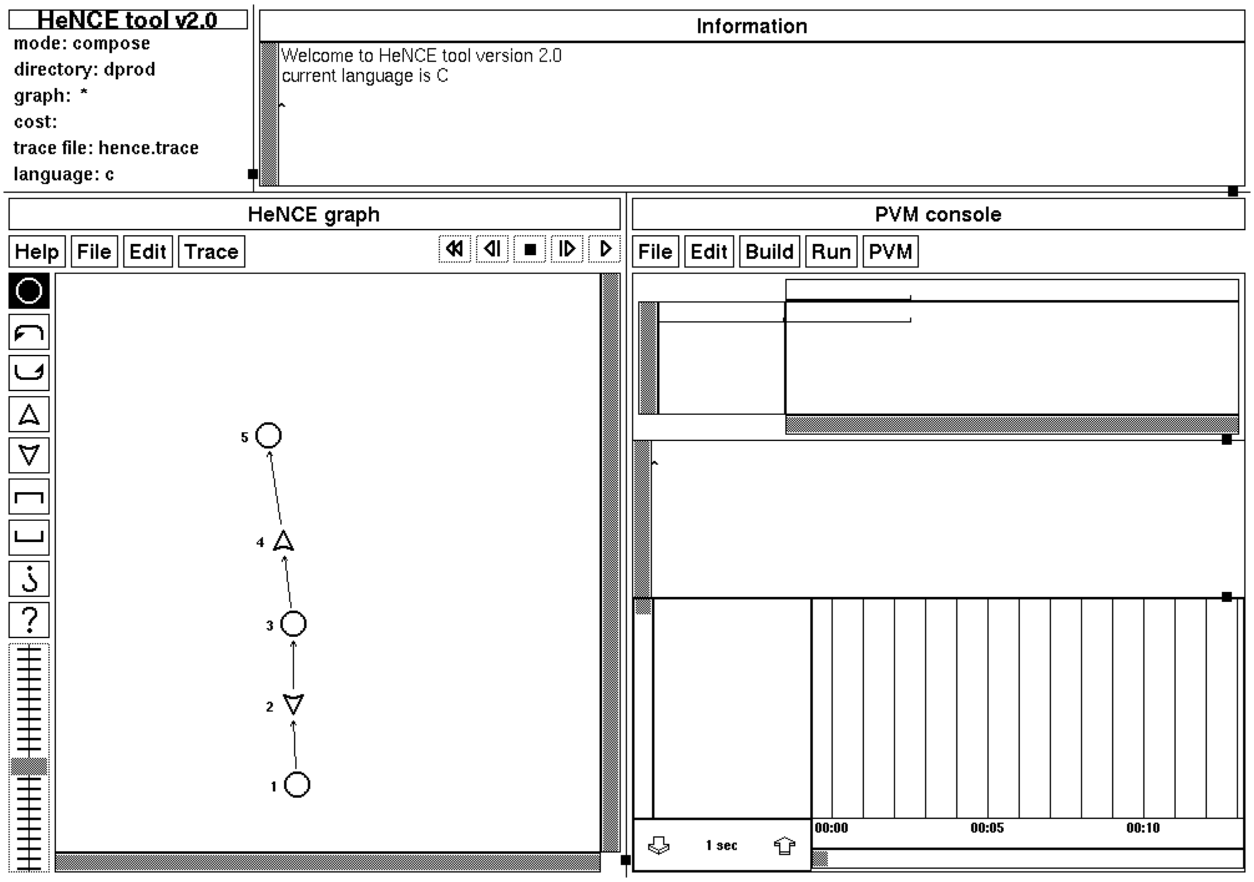


Figure 10: Dot Product HeNCE Graph

Move the mouse to the lower center of the htool graph drawing area and click the left mouse button. A Computation node should appear.

Now draw the next node up from the bottom. Click (left) on the Begin Fan node icon to select and then click (left) above the Computation node just drawn. The Begin Fan should appear. Continue this procedure for all of the nodes in the graph.

4. Next draw the four arcs in the graph. Notice that they all point up. To draw an arc, first move the mouse to the "from" node and press and hold the center mouse button. Continuing to hold the mouse button pressed, move the mouse to the "to" node and release the mouse button. This should draw the arc. Do this for all four arcs.

5. At this point, you are ready to enter each node's program as well as inspect the node subroutines that have already been added.

   Begin with the bottom node (labeled 1 in Figure 10). Open its node program by clicking the right mouse button on it. At this point, an editor window with text like that shown in Figure 11 will appear.

   You should ignore the first line of this text. Do not edit it. However, you must replace the UNDEFINED line and (optionally) the comment with the desired node program. Edit the text to read like that shown in Figure 12, but remember not to change the first line.

   In this node program, the lines that begin with NEW declare variables and allocate storage for them. Many of the variables are also given initial values. The meaning within the program of the variables

```
NODE [ 307 116 ] 1
/* replace the UNDEFINED keyword with the node program */
/*
 * Format: any input-only or input-output declarations go first, e.g:
 *    NEW <> foo[100];
 * followed by a subr call, e.g:
 *    x = xyzzy (a, b);
 * then any output-only declarations:
 *    > bar[42];
 */
    UNDEFINED;
```

Figure 11: Node Program Displayed for New Computation Node

```
NODE [ 307 116 ] 1
    NEW <> int n=100;
    NEW <> int nproc=10;
    NEW <> float sum[nproc];
    NEW <> float x[n];
    NEW <> float y[n];
     init(x, y, n);
```

Figure 12: Node Program for Node Init

is described in Figure 9. The line containing "init" is the call to sequential subroutine `init` which is defined in file "init.c" and gives initial values to vectors x and y. When this node fires, `init` will be called.

When you close the editor window, the text you entered will be parsed. If there is an error, htool will complain and give you a chance to make changes. Otherwise, the subroutine name will appear next to the node.

You can look at subroutine `init` at this point by shift-clicking the right mouse button on the node. The subroutine will appear in a text edit window. If file "init.c" did not already exist, you would have to type it in at this point. You can inspect the node subroutine of the other compute nodes the same way, after their node programs have been entered.

Next, you must give the Begin Fan node (labelled 2) its node program as shown in Figure 13. Do not change the FANOUT line, but instead replace the rest with the "p =" line shown. This means that the fan will create `nproc` copies of the subgraph (consisting of one compute node) within it. Each of the copies will have access to a separate variable `p` which will have value that range from 0 to nproc-1, inclusive.

Next, give the compute node labelled 3 the node program shown in Figure 14. This is the node that is replicted by the fan. All copies of it will run in parallel. The call to `dprod` will compute the dot product of vectors `a` and `b` which are subvectors of `x` and `y`. The lines with the word "NEW" define these

```
FANOUT [ 182 306 ] 2
    p = 0 TO nproc-1;
```

Figure 13: Node Program for Begin Fan

```
NODE [ 174 247 ] 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 14: Node Program for Node dproc

```
NODE [ 174 116 ] 5
    < int nproc;
    < float sum[];
     output(sum, nproc);
```

Figure 15: Node Program for Node output

subvectors. Notice that this program will work only when nproc evenly divides n. More complicated expressions would be needed for the general case.

The line "<> sum[p]" is interesting. The "<>" means that what follows will be read from a predecessor node and that any changes made to it will be available to successor nodes. So, each copy of this node will make use of a single element `p` of vector sum defined in node `init` to store the result of its dot product. The elements of sum used by each copy of this `dproc` node will be available as vector `sum` to node `output`.

No control construct ending node requires a node program so there is nothing to be done for the End Fan node with label 4, but you must give the top compute node (label 5) the node program shown in Figure 15. The "<" lines mean that this node gets `nproc` and `sum` from the nearest predecessor node that defines them. Im this program, `nproc` comes from `init` and the elements of `sum` are set by the copies of the `dproc nodes`. Subroutine `output` totals the values in `sum` and prints the result.

6. At this point, you should save the graph that you have drawn and annotated. Select "Save" from the File menu. You will have to move the mouse over the file name box and enter text. Graph file names end in ".gr" by convention so "dprod.gr" is a good name to choose.

7. You are now ready to try and build and run the program. First try to run it on the host on which you ran htool. This is the default if you do not enter a cost matrix. Pick "Build" from the PVM Console Build menu. Watch the htool window for error messages. If there are none, you can run your program by selecting "Run Program" from the PVM Console Run menu. You should see animation on the HeNCE graph as the program runs.

   When the program completes, a message "HeNCE executioner exited normally" will appear in the PVM Console. At this time, pick "Exit Run Mode" from the PVM Console Run menu. Notice that HeNCE has created a cost matrix for you. It contains only one host, the one on which you are running htool.

8. Now try to run the dot product program using two machines. For this, you will have to add another host to the cost matrix. This must be a host that you can rlogin to without entering a password and for which PVM and the HeNCE libraries have been installed.

   Pick "Add from Graph" from the PVM Console Edit window. Another row will be added to the default cost matrix created in the previous step. You must enter values into all of its fields by moving the mouse to a field and typing. Enter the hostname of the second machine and costs for all subroutines. You can just use the value 1 for all costs. At this point, save the cost matrix by picking "Save" from the PVM Console File menu and entering a file name, which should end in ".mat" by convention.

Build and run just as before. You will see animation on the graph and two hosts in the strip chart at the bottom of the PVM Console. When the program completes, pick "Exit Run Mode" mode also just like before.

9. When a HeNCE program runs, it creates a trace file which you can use to review your programs execution. Pick "Load Trace File" from the HeNCE Graph Trace menu. Notice that HeNCE has created a trace file with the default name "hence.trace." Go ahead and load it to enter trace mode. The VCR-like control icons in the upper right of the graph segment control the replay. Click on the rightmost, the "play" button. You will see the runtime animation repeated. Press the left-most button to rewind. Then repeatedly press the button that is second from the right. This is the single step forward button. The middle button is "stop," and the second button is "single step backwards." When you have finished experimenting with trace replay, pick "Exit Trace Mode" from the Trace menu.

10. This is the end of the tutorial. You have entered and run a parallel program using more than one host, and you have replayed its execution using HeNCE's trace facility.

    Before you leave htool, pick "Halt PVM" from the PVM Console PVM menu. Do not forget to do this, or you will leave PVM daemon processes running. To exit htool, pick "Quit" from the HeNCE Menu at the upper left of the htool window.

11 Customizing HeNCE

There are two mechanisms by which htool's operation and appearance may be customized. X windows related attributes such as colors and fonts may be defined as X resources. Other attributes may be set in a ".htool" file. Most users will have no need to customize htool using either mechanism since all important values have reasonable defaults on most common architectures.

11.1 Using a .htool File

Many HeNCE attributes may be set by adding lines to a file called ".htool" which must be placed in your home directory. Each line contains a variable name and a value and should have the following form.

```
NAME: Value
```

A list of all of the variable names that can be set follows. Notice that all variable names must be in upper case. Don't forget the colon.

1. EDITOR_COMMAND (default: xterm -title %s -e vi %s)

   The command htool uses when you want to edit a node program or node subroutine. Emacs users can set it to "emacs %s" since emacs can create its own window when running under X.

2. LANGUAGE (default: C)

   The language to be used for your node subroutines. The value musty be either C or FORTRAN.

3. PRINT_COMMAND (default: lpr %s)

   The command HeNCE will use to print a Postscript image of your graph.

4. TMP_DIR (default: /tmp)

   The UNIX directory where htool scribbles temporary cost matrices, etc.

5. PVM_HOST_FILE (none by default)

   Name of PVM host file to be used when pvm3d is started.

6. SUB_DEFS_FILE (none by default)

   Name of file from which htool will read function prototypes for node subroutines. Htool automatically determines this information from the actual parameters passed in node programs. This facility allows users to override htool's type determination. It is unlikely that you will need to use this feature.

7. ARCH_CC (default depends on architecture)

   ARCH is a PVM architecture name such as SUN4, RS6K, or HPPA. See the section on supported hardware.

   This variable is a printf-like string that says how to compile a C node program. Special escapes include the following:

```
%{something}     include "something" only if debugging
%[something]     include "something" only if NOT debugging
%D               represents predefined ``-D'' preprocessor defines
%I               represents predfined ``-I'' includes
%L               represents predefined ``-L'' and ``-l'' libraries
%o               name of object file (foo)
%s               name of source file (foo.c or foo.f)
%w               name of wrapper (cw_foo.c or fw_foo.f)
%W               name of compiled wrapper (cw_foo.o or fw_foo.o)
```

   Htool has special wired in values for HPPA, SUN4, PMAX, and RS6K, which can be overridden here. For other architectures, the following default is used.

```
cc -o %o %{-g}%[-O] %D %I %s %w %L -lm
```

   If you wish to create a custom ARCH_CC, you do it by adding to the basic string. You should continue to use the escapes that represent the items that ae known to be reequired. For example, to add a new library, simple add a new "-l" to the basic string as shown below.

```
cc -o %o %{-g}%[-O] %D %I %s %w %L -lm -lsomenewlib
```

8. ARCH_FC (default depends on architecture)

   A printf-like string that says how to compile a FORTRAN node program. Escapes are the same as for ARCH_CC. ARCH is an architeture name. If not otherwise defined, the following is assumed:

```
cc %{-g}%[-O] %D %I -c %w && f77 -o %o %{-g}%[-O] fmain.f %W %s %L -lm
```

   There are defaults for HPPA, SUN4, PMAX, and RS6K.

9. ARCH_BUILD_FLAGS (depends on architecture).

   ARCH is an architecture name. The BUILD_FLAGS are a string which contains any of the following characters:

```
_   append an underscore to function name to call from FORTRAN
U   uppercase function names to call from FORTRAN
l   add extra args at the end of arg lists to give the length
    of FORTRAN CHARACTER arrays
C   special CRAY hacks for fortran strings (unimplemented)
T   special TITAN hacks for fortran strings (unimplemented)
```

Resource	Example value	Description
titleBackground	cyan	background color for title bars
titleForeground	black	foreground color for title bars
brokenColor	red	Indicates one of: build failed, can't configure host, host died, or proxyd died
deadColor	black	Indicates node program finished entirely and exited
doneColor	magenta	Indicates build finished compiling or node has completed execution
errorColor	red	Indicates an error in a node program
idleColor	yellow	Indicates a compilation in progress or a host, node, or subroutine that is waiting for requests from children
normalColor	white	Default background color for cost matrix
notReadyColor	grey	Indicates node not ready to execute
readyColor	yellow	Indicates node ready to execute
runningColor	green	Indicates a build in progress or a host, node, or subroutine that is currently running
warningColor	orange	Indicates that there is a warning about the HeNCE graph for a particular node

Table 3: Htool X resources.

There are defaults for HPPA, SUN4, PMAX, and RS6K. No flags are automatically included for other architectures.

Just to summarize, here is a small but complete example of a .htool file. It sets both the langauge for node programs and the editor to be used.

```
EDITOR_COMMAND: xterm -title %s -e mg %s
LANGUAGE: FORTRAN
```

11.2 X Resources

Since htool is an X-based program, its behavior can be customized by setting X resources. X resources are set by placing a line in your .Xresources file and typing the command `xrdb -load .Xresources`. You may want to add the "xrdb" command to your ".xinitrc" file to have it run automatically when you start X. The htool resources and their defaults are listed in Table 3. Some example lines that you could put into your ".Xresources" file follow. They cause most things to be drawn in blue.

```
htool.notReadyColor: blue
htool.readyColor: blue
htool.doneColor: blue
htool.runningColor: blue
```

The resources `brokenColor`, `deadColor`, `doneColor`, `errorColor`, `idleColor`, `normalColor`, `notReadyColor`, `readyColor`, `runningColor`, and `warningColor` are used to specify colors of the nodes when drawing graphs in compose and trace mode.

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.

12 Non-Standard PVM Environments

Before executing a HeNCE program, htool will start a virtual machine. PVM will be started if necessary and any hosts defined in the cost matrix will be added to the machine's configuration. By default, PVM accomplishes this using the system's *remote shell* facility (`rsh` command on most machines; `remsh` on a few others). PVM normally expects to use the same username on the remote machine as on the local machine and expects you to be able to rlogin to all machines in the virtual parallel machine without entering a password.

If this is not the case, the best approach is probably to start PVM manually and add all hosts in your virtual parallel machine before starting htool. You accomplish this by starting a pvm daemon (pvmd3) in the foreground on the machine on which you will run htool. You can supply pvmd3 with a PVM host file. There are options to use alternative usernames and be prompted for passwords. See the PVM 3.X User's Guide for information on host files.

It is possible to have htool start PVM with a hosts file by setting the PVM_HOST_FILE variable in your .htool file. However, the password option will not work. Other host file options will likely work.

13 Building HeNCE from Source

The source code to htool and the HeNCE execution libraries comes in the same package. In order to build htool, your workstation must have X windows version X11R4 or X11R5 and the Athena Widget Libraries installed.

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. The X window system may be obtained via anonymous ftp from x.org.

13.1 Quick Instructions

Before compiling HeNCE, you must have installed PVM 3.2.4 or later in directory pvm3 in your home directory.

If you are lucky, your system has a properly installed X windows environment along with a correctly configured "imake" facility. Unfortunately, many workstation vendors fail to provide this. This is the major reason that we distribute HeNCE in pre-compiled form.

If all is well on your system, you can build and install htool and its execution environment by following these instructions. The final install step will place everything within the pvm3 directory in your home directory. Attempts to place it elsewhere are discouraged.

1. Unpack the hence2.0 package anywhere and cd to the hence2 directory. Be sure your X11 bin directory is on your path.
2. Enter "xmkmf".
3. Enter "make Makefiles", noting that uppercase "M" and the "s" on the end.
4. If you have old object (.o) files left over from a previous HeNCE compilation on a different system, enter "make clean" to remove them.
5. Enter "make"
6. Enter "make install"

If you just wish to build and install the HeNCE execution support software (probably because you have already installed htool for some other machine type), the process is not dependent on correctly installed X software. Hence, it is much more likely to be trouble-free. Again, PVM must be built and installed in pvm3 within your home directory. Follow these steps.

1. Unpack the hence2.0 package anywhere (if you have not already done this) and cd to the hence2 directory.

2. Enter "make -f Makefile.lib".

3. Enter "make -f Makefile.lib install" to copy the runtime libraries into the pvm3 directory.

13.2 Detailed Instructions for Building htool

HeNCE uses the X window system's "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.

If you are building HeNCE for a new system (one which HeNCE does not already support), you will need to add a new section to the file named `imake.defs` to recognize your machine. You can do so either using symbols defined by your system's imake config files (usually in `/usr/lib/X11/config`), or symbols defined by your system's C preprocessor. The section that pertains to your machine needs to set the `PVM_ARCH` variable to match the one used by PVM on your machine. It may also want to set `CCOPTIONS`, `HENCE_DEFINES`, `HENCE_LIBRARIES`, or `LOCAL_LDFLAGS` if needed to override the defaults on your machine. Do **not** change the locations of `PVM_ROOT`, `PVM_LIBDIR`, `LIBPVM`, `PVM_INCLUDEDIR`, or `BINDIR`. HeNCE expects these locations to be the same on every machine.

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`).

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 into directory pvm3 in your home dircetory.

Earlier versions of HeNCE have also been used on Cray, NeXT, Sequent, and Silicon Graphics 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.

14 Trouble Shooting

If you are having trouble using HeNCE, you may want to check some of these following points.

1. Is environemt PVM_PATH set automatically to "~/pvm3" when you login to any of the hosts in your virtual parallel machine? You must do this in your .cshrc or .profile (depending on your login shell) if you are using PVM 3.3 or later. Users of ksh might have troubles with this and should consult PVM's documentation on how to set PVM_PATH.

2. Is PVM installed in "~/pvm3?"

3. Are you running pvm version 3.2.4 or later? You can check by running the pvm console "~/pvm3/lib/pvm" and entering "version" followed by "halt" to exit the console.

4. Is "~/pvm3/bin/ARCH" in your path for every host in your virtual parallel machine. "ARCH" is a PVM architecture name.

5. Can you rlogin from your htool host to all hosts in your virtual parallel machine without entering a password? If not, see "The Default PVM Environment" above.

6. Does PVM work by itself? You can test it by running one its example programs.

7. Did you compile HeNCE with the same version of PVM that is installed in "~/pvm3?"

8. The binary distribution of HeNCE for Sun 4 machines assumes that you use the openwindows libraries as distributed with SunOS 4.1.3 (Solaris 1.1). You may have to set environment variable LD_LIBRARY_PATH to /usr/openwin/lib if htool complains about a missing library.

```
setenv LD_LIBRARY_PATH /usr/openwin/lib
```

15 Known Bugs and Limitations

1. Htool cannot effectively use the password option in a PVM host file. If you need it, start PVM manually and add all hosts (with a host file) before running htool.

2. We have had problems with HeNCE execution failing on DEC PMAX (not Alpha) machines with all versions of PVM.

3. HeNCE does not work with PVM 3.3 on DEC Alpha machines running PVM 3.3 under OSF/1 V1.2. HeNCE does appear to work with PVM 3.2.6 on this platform.

4. There is a problem such that data written to standard output fails to appear in the htool output window. You can work around this by explicitly flushing standard output and then having the final node in your program delay for a few seconds. In "C" you would use fflush(stdout) and sleep(5), for example.

5. Htool does not warn you if you fail to save a graph or cost matrix.

16 Acknowedgements

HeNCE version 2.0 benefits greatly from those who worked on previous versions. Jim Plank and Reed Wade deserve special mention. Thanks also to pre-release users and all others who send in suggestions and comments.

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.

[3] J. J. Dongarra, T. Rowan, R. Wade. Software distribution using XNETLIB. ACM Transactions on Mathematical Software, (to appear) 1994.