Chapter 10

Node Prism

Node Prism is the version of Prism used for debugging, analyzing the performance of, and visualizing data in CM-5 message-passing (MIMD) programs (that is, programs that use the CMMD library). This chapter describes how to use Node Prism; it assumes that you are already familiar with Prism when used with data parallel or scalar programs. (If you aren't familiar with Prism, the chapter points you to the appropriate sections in previous chapters.)

Knowledge of **`pndbx`**, our previous product for debugging message-passing programs, is helpful but not necessary for understanding the material in this chapter. Some of the commands discussed here are similar to **`pndbx`** commands, but the syntax is often changed or expanded in Node Prism.

In this chapter, we use the term *Global Prism* to refer to the version of Prism discussed in previous chapters of this manual. Global Prism is used to debug data parallel and scalar programs that don't use the message-passing model (this includes the host program in the host/node version of the message-passing model).

10.1 Overview

In message-passing programs, each individual node runs its own copy of a program. At a given point in the execution of the program, all nodes may be doing the same thing, or (more likely) different subsets of nodes may be doing different things.

To debug such a program, you may want to look at the behavior of individual nodes or of particular groups of nodes. A key concept in Node Prism is the *PN set*; a PN set is a predefined or user-defined group of nodes that you can

Copyright © 1991–1994 Thinking Machines Corporation

view and operate on as a single entity. Aggregating nodes into PN sets can help provide you with the correct level of analysis for your program.

In general, Node Prism operates in much the same way as Global Prism. The differences in many cases involve the application of the concept of PN sets to Global Prism features such as events and visualizers.

To learn:

- **How to enter Node Prism,** see Section 10.2. You can either enter Node Prism directly, by using the **`-node`** option to the **`prism`** command, or via the **PN Debug** selection from the **Utilities** menu in Global Prism.
- **How to create and use PN sets,** see Section 10.3. Node Prism provides a variety of ways of creating and deleting PN sets. You can use PN sets to qualify many Prism commands, so that the command applies only to the nodes that are members of the PN set.
- **How to execute programs in Node Prism,** see Section 10.4. Node Prism lets you interrupt execution on individual nodes or sets of nodes, and wait for execution to stop in the nodes of a PN set.
- **How to debug in Node Prism,** see Section 10.5. In Node Prism, you can create events that are tied to specified PN sets, so that, for example, you can set a breakpoint only on certain nodes. Node Prism also provides a **Where** graph that displays a snapshot of the dynamic call graph of the program, for nodes that aren't running.
- **How to visualize data in Node Prism,** see Section 10.6. Node Prism adds an extra "node" axis to visualizers, so that you can see the values of a variable on each node in a PN set.
- **How to analyze performance in Node Prism,** see Section 10.7. Node Prism provides separate performance data for each node on which a message-passing C* or CM Fortran program is running.
- **About customizing Node Prism,** see Section 10.8.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

10.2 Entering Node Prism

See Section 2.2 for basic information about entering Prism.

10.2.1 From the Command Line

To enter Node Prism directly from the command line, issue the **prism** command with the **-node** option:

```
% prism -node
```

As with Global Prism, you can specify the name of an executable program on the **prism** command line, and Prism will come up with that program loaded. You can also specify the **-C** option to bring up Node Prism in commands-only mode, or the **-CX** option (from an Xterm) to bring it up in commands-only mode, but be able to send the output of certain commands to X windows.

In addition, you can associate a node core file with the executable program that you load. Add the name of the core file after the name of the program. This can give you some information about the state of this node. For instance, you can issue a **where** command to get the stack trace, and you can print ordinary variables that live in static memory or stack memory. However, you can't view variables living in vector-unit memory, nor can you look at data from other nodes. See the *CM-5 User's Guide* for more information about CM-5 core files.

Node Prism's main window is virtually the same as Global Prism's. There are two differences:

- There are fields for **Current PN Set** and **Current PN** in the status region above the source window. These fields are discussed in Sections 10.3.6 and 10.3.7.
- The command-line prompt is different. Once again, see Section 10.3.6.

Copyright © 1991–1994 Thinking Machines Corporation

10.2.2 From Global Prism

Some message-passing programs may have a host program that runs on the partition manager. Node Prism doesn't provide a mechanism for debugging or analyzing the performance of a host program. To do this, you can simply start up a Global Prism (without the **-node** option) and load the host program into it; you can then work on the host program as you would any scalar program.

If you have started up Global Prism and loaded a host program, you can then start up a commands-only version of Node Prism on the corresponding node program by clicking on **PN Debug** in the **Utilities** menu.

See Section 10.4.1 for another method of using Global and Node Prism on the host and node parts of a host/node message-passing program.

10.2.3 From a Remote Computer

You can execute Node Prism on a computer that is linked via a network to the CM-5 on which your message-passing program is to be run; Prism does not have to execute on a partition manager of this CM-5. This is known as *remote execution*.

The advantage of executing Node Prism on another computer is that it reduces the load on the partition manager. The disadvantage is that heavy network traffic could cause some degradation of Prism's performance.

The computer on which you run Node Prism must be a Sun workstation running SunOS 4.1.1 or later.

Opening the Connection

You begin by logging in to the computer on which you want to run Node Prism's user interface. Then execute the **prism** command with the **-connect** option; as its argument, specify the name of the partition manager from which you want your program to run. For example, if you want your program to run on partition manager Ernie, issue the command

```
% prism -node -connect ernie
```

(You can also specify any other **prism** command-line options.)

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

When graphical Prism starts up, the name of the partition manager appears in its title bar.

If you don't use the **-connect** option, you can issue the **connect** command on the Prism command line, once again specifying the name of the partition manager as its argument. For example,

```
connect ernie
```

You can also specify the keyword **manual** when you issue the **connect** command. (Include it after the name of the partition manager.) This tells Prism to print the command it would use to start the remote process on the partition manager, rather than execute the command itself. The command is printed in the Xterm from which you started Prism. It would look something like this:

```
run on ernie: /users/cmsg3/bowker/sde/src/main/prism_mimd_server\
    -rh dickens.think.com -rp 1540
```

You can then log in to the partition manager (using whatever logname or access codes are required) and execute the command yourself. This is useful if access to the partition manager is restricted for any reason, so that Prism cannot execute the command.

After printing the command, Prism waits until it is executed. In commands-only Prism (but not graphical Prism) you can type **Ctrl-c** to interrupt the wait.

Note these points in connecting to a partition manager:

- You can have only one remote session active at a time. That is, you cannot issue multiple **connect** commands to connect to different partition managers.
- Once you have connected to a partition manager, you cannot use Prism to debug a program running on the computer from which you are running Prism. You can only debug a program running on the partition manager.
- The name of the process running on the partition manager is **prism_mimd_server**.

Copyright © 1991–1994 Thinking Machines Corporation

Closing the Connection

To close the connection to a remote partition manager, issue the command **disconnect**. No options are needed, since only one connection can be open.

Mapping Paths

It is possible that files and directories will have different paths on the local computer and the remote computer. If this is the case, you can use the **pathmap** command to provide a mapping between the local and remote paths. The syntax is

pathmap *local_path remote_path*

where *local_path* is the pathname on the computer on which you are running Prism, and *remote_path* is the corresponding pathname as viewed from the partition manager to which Prism is connected.

For example, if the local path appears as **/users/csg1** and the corresponding path, when viewed from the partition manager, is **/cpmount/users/csg1**, issue the command

```
pathmap /users/csg1 /cpmount/users/csg1
```

If you then load the file **/users/csg1/sistare/primes.x** and issue the **run** command, Prism will look for the file **/cpmount/users/csg1/sistare/primes.x** to execute on the partition manager.

The mapping is also used in the **cd** command. For example, if you issue the **pathmap** command shown above and then issue this command

```
cd /users/csg1/sistare/bin
```

the current directory on the partition manager will be **/cpmount/users/csg1/sistare/bin**.

You can issue multiple **pathmap** commands, providing multiple mappings. Prism searches through them in order until it finds a match. If it doesn't find a match, it doesn't change the local path on the partition manager. This could mean, of course, that it won't find the executable program when you try to load it.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

To avoid having to specify these mappings during each Prism session, you can use the Prism X resource **Prism.pathMap** to list the mappings; see Section 9.4.11. Note, however, that commands-only Prism is not aware of the settings of Prism resources unless the settings are contained in the system-wide Prism **`app-defaults`** file.

To show all current mappings (including those specified via the **Prism.pathMap** resource), issue **`pathmap`** with no arguments.

To delete all current mappings, issue the command

```
pathmap delete
```

10.3 Using PN Sets

Node Prism provides you with the capability of viewing your message-passing program at the level of an individual processing node (PN), a group of nodes, or all nodes on which it is executing. For example, at times it may be useful to look at the status of node 0, because you have reason to believe there is a problem with it. At other times you may want to look at all nodes that have encountered an error, or at all nodes in which the value of a particular variable exceeds a certain number.

These groups of nodes, typically chosen because they have some useful characteristic in common, are referred to as *PN sets*. Node Prism treats a PN set as a unit; for example, you can use the name of a PN set as a qualifier for many commands. The command is then executed for each node in the set. For example, you can set a breakpoint that applies only to nodes in a specified PN set. (See Section 10.3.9 for more information.) In addition, many graphical actions apply only to nodes in a PN set.

If you don't need to view your program at the level of an individual node or a subset of nodes, you can also view its operation on all the nodes in your partition.

You can view PN sets in the **PN Sets** window, as described in Sections 10.3.1 and 10.3.4.

As we describe in Section 10.3.2, Prism provides predefined PN sets for certain standard groups of nodes; for example, the set of all nodes in an error state is a predefined PN set. You can also define your own PN sets, as

Copyright © 1991–1994 Thinking Machines Corporation

described in Section 10.3.3; for example, you can define a PN set to be those nodes where **x** is greater than 0. Section 10.3.5 describes how to delete PN sets.

If you don't specify a PN set as a qualifier to a command (that can take a PN-set qualifier), the command is executed on the *current* PN set; many graphical actions also apply to nodes in the current set. The concept of the current PN set is described in Section 10.3.6. Section 10.3.7 describes the current PN, which is a distinguished node within a PN set.

Section 10.3.8 describes the **cycle** PN set, which is a predefined PN set with special characteristics.

10.3.1 Using the PN Sets Window

You can use the **PN Sets** window to view the current status of the nodes in your partition and to perform many of the actions associated with PN sets.

To display the window:

- **From the menu bar:** Choose the **PN Sets** selection from the **Debug** menu.
- **From the command window:** Issue the command **show pnsets on dedicated**.

Figure 40 shows the **PN Sets** window for a 32-node CM-5 partition.

The various components of the window are described in detail in later sections. Here is a brief overview:

- The main area of the window shows PN sets and their members. Nodes that are members of a set are shown as black (or colored) cells within a square that represents the entire partition on which the program is running. The nodes are numbered starting at the upper left, increasing from left to right and then jumping to the next row. Shift-click on a cell to view its node number. The current PN (see Section 10.3.7) for each PN set is shown in gray (or, on a color workstation, a darker shade of the color in the other squares). The current PN set (see Section 10.3.6) is shown in the upper left corner of the window.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

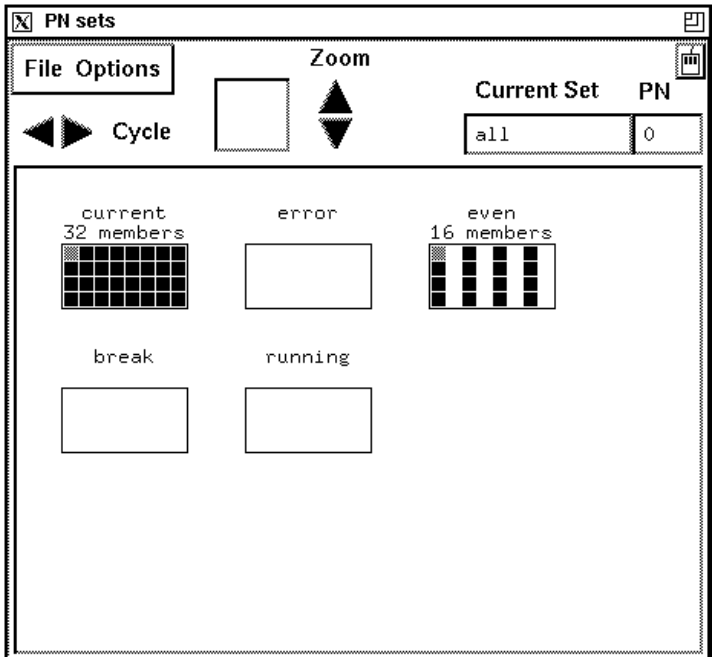

Figure 40. The PN Sets window.

- You can cycle through the **`cycle`** PN set (see Section 10.3.8) by clicking on the left and right arrows labeled **Cycle** at the top left of the control panel.
- If you have many PN sets and a large partition, you can use the **Zoom** arrows to zoom in or out on these PN sets. The box next to the arrows shows what part of the entire display you are seeing; you can drag the mouse through this box to pan through the display.
- You can view and change the current PN set and current PN via the boxes at the top right of the window
- The **Options** menu at the top left of the window lets you hide, display, create, and delete PN sets. See Sections 10.3.3 through 10.3.5.
- The **File** menu lets you close the **PN Sets** window.

Copyright © 1991–1994 Thinking Machines Corporation

10.3.2 Predefined PN Sets

Node Prism provides the predefined PN sets described below.

- **`all`** — This set contains all the PNs in the CM-5 partition (in other words, all the nodes on which the program is executing). This is the default PN set.
- **`running`** — This set contains all PNs on which the program is currently executing.
- **`error`** — This set contains all PNs that have encountered an error.
- **`interrupted`** — This set contains the PNs that were most recently forcibly interrupted by the user. See Section 10.4.8 for a discussion of the **`interrupt`** command and a further explanation of this PN set.
- **`break`** — This set contains the PNs that are currently stopped at breakpoints.
- **`stopped`** — This set contains all PNs that are currently stopped. It is the union of the sets **`error`**, **`interrupted`**, and **`break`**.
- **`done`** — This set contains all PNs that have terminated successfully.

Except for the PN set **`all`**, these sets are *dynamic*; that is, as a program executes, Prism automatically adjusts the contents of each set to reflect the program's current state.

In addition, there are two set names that have special meaning in Node Prism: **`current`** and **`cycle`**. They are discussed in Sections 10.3.6 and 10.3.8, respectively.

10.3.3 Defining Your Own PN Sets

The Syntax for Defining a PN Set

This section describes the syntax you can use to specify a PN set. As described below, you can assign a name to a PN set you specify using this syntax; this provides a useful shorthand for complicated PN set specifications.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

First, let's look at the syntax you use to specify a PN set as an argument to a command:

To apply a command to a specific PN set, use the **pnset** keyword, followed by a PN specification. Put this **pnset** clause at the end of the command line (but before an **on** *window* clause, if any). Thus,

```
print x pnset error
```

prints the values of the variable **x** in the predefined PN set **error**. (See Section 10.6 for a discussion of printing variables in Node Prism.)

Now let's look at the ways in which you can specify your own PN set as part of this **pnset** clause:

Specify an individual PN number. An individual PN can constitute a PN set. Thus,

```
print x pnset 0
```

prints the value of **x** in node 0.

Specify the name of a PN set. You name a PN set using the **define pnset** command, as described below. Thus,

```
print x pnset foo
```

prints **x** in the nodes you have defined to be members of PN set **foo**.

Specify a list of PN numbers. Separate the numbers with commas. Thus,

```
print x pnset 0, 4, 7
```

prints **x** in nodes 0, 4, and 7.

Ranges and strides are allowed. Use a colon between two node numbers to indicate a range. Use a second colon to indicate the stride to be used within this range. Thus,

```
print x pnset 0:10
```

prints **x** in nodes 0 through 10. And

```
print x pnset 0:10:2
```


Copyright © 1991–1994 Thinking Machines Corporation

prints **x** in nodes 0, 2, 4, 6, 8, and 10.

You can combine comma-separated node numbers and range specifications. For example,

```
print x pnset 0, 1, 3:5, 8
```

prints **x** in nodes 0, 1, 3, 4, 5, and 8.

Specify a union, difference, or intersection of PN sets. To specify the union of two PN sets, use the symbol **+**, **|**, or **||**. For example,

```
print x pnset 0:2 + 8:10
```

prints **x** in nodes 0, 1, 2, 8, 9, and 10.

```
print x pnset foo | bar
```

prints **x** in nodes that are members of either PN set **foo** or PN set **bar**.

Prism evaluates the PN set expression from left to right. If a node returns **true** for the first part of the expression, it is not evaluated further. In the above example, if a node is a member of **foo**, its value of **x** is printed; Prism does not check its membership in **bar**.

Use a minus sign to specify the difference of two PN sets. For example,

```
print x pnset stopped - foo
```

prints **x** in all nodes that are stopped except those belonging to the PN set **foo**.

To specify the intersection of two PN sets, use the **&**, **&&**, or ***** symbol. For example,

```
print x pnset foo & bar
```

prints **x** in nodes that are members of both PN set **foo** and PN set **bar**. Prism evaluates the PN set expression from left to right. If a node returns **false** for the first part of the expression, it is not evaluated further. In the above example, if a node is not a member of **foo**, Prism doesn't bother checking its membership in **bar**; it won't be printed in any case.

Prism must evaluate a PN set expression in each node at the time the command is executed; the nodes must be stopped for Prism to do this. The evaluation fails if any of the nodes being evaluated are running. Using the

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

predefined PN set **stopped** on the left of an intersection expression is a useful way of ensuring that a command applies only to stopped nodes. Thus,

```
print x pnset stopped & foo
```

prints **x** only in the members of **foo** that are stopped.

Specify a condition to be met. Put braces around an expression that evaluates to true or false on each node. Nodes in which the expression is true are part of the set. Thus,

```
print x pnset { y > 1 }
```

prints **x** in nodes where **y** is greater than 1. And

```
print x pnset all - { y == 1 }
```

prints **x** in all nodes except those in which **y** is equal to 1.

Membership in a set defined with this syntax can change based on the current state of your program; such a PN set is referred to as *variable*. See the section "Evaluating Variable PN Sets," below, to learn how to update the membership of a variable PN set.

For this syntax to work, the variable must be active in all nodes on which the expression is evaluated. If the variable isn't active on a node, you get an error message and the command is not executed. To ensure that the command is executed, use the intrinsic **isactive** in the PN set definition. The expression **isactive(***variable***)** returns **true** if *variable* is on the stack for a node or is a global. Thus, you could use this syntax to ensure that **x** is printed:

```
print x pnset stopped && {isactive(x)}
```

Naming PN Sets

You can assign a name to a PN set. This is convenient if you plan to use the set frequently in your Prism session.

Use the syntax described above to specify the PN set. You can use any name except the names that Prism predefines; see Section 10.3.2. The name must begin with a letter; it can contain any alphanumeric character, plus the dollar sign (**$**) and underscore (**_**).

Copyright © 1991–1994 Thinking Machines Corporation

From the PN Sets window: Choose **Define** from the **Options** menu. A dialog box is displayed that prompts for the name and definition of the PN set. Click on **Create** to create the PN set.

From the command line: Issue the **`define pnset`** command.

For example,

```
define pnset odd 1:31:2
```

creates a PN set called **`odd`** containing the odd-numbered nodes between 1 and 31.

```
define pnset xon { x .NE. 0 }
```

defines a PN set consisting of those nodes where **`x`** is not equal to 0. Note that **`x`** must be active in all nodes for this syntax to work. As described above, you can use the intrinsic **`isactive`** to ensure that **`x`** is active on the nodes that are evaluated. For example,

```
define pnset xon { isactive(x) && (x .NE. 0) }
```

Both versions create a variable PN set whose contents will changed based on the value of **`x`**. See below for more discussion of variable PN sets. Finally, note that all nodes must be stopped for this syntax to work. To ensure that the definition applies only to stopped nodes, use this syntax:

```
define pnset xon stopped && { isactive(x) && (x .NE. 0) }
```

User-defined PN sets are deleted when you reload a program. To get a list of these PN sets before reloading, issue the command **`show pnsets`**. You can then use this list to help reissue the **`define pnset`** commands. See Section 10.3.4 for more information about **`show pnsets`**.

Evaluating Variable PN Sets

We have already discussed how to create variable PN sets — sets whose contents can change as the program executes. Prism evaluates the membership of such a set when it is defined. If no nodes meet the condition (for example, because the program is not active), Prism prints appropriate error messages, but the set is defined.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

Subsequently, you can re-evaluate the membership of the PN set by issuing the **eval pnset** command, specifying the name of the PN set as its argument. For example,

```
eval pnset xon
```

evaluates the membership of the PN set **xon**. This causes the display for the PN set to be updated in the **PN Sets** window.

Note that this evaluation will fail if:

- nodes are running that need to be polled in evaluating the PN set; or
- the PN set's definition contains a variable that is not active on any of the nodes being polled

For example, if you issue this command:

```
define pnset foo { x > 0 }
```

you must make sure that all nodes are stopped, and **x** is active on all nodes, when you issue the command

```
eval pnset foo
```

To ensure that the evaluation succeeds, you would need to use the more complicated syntax described above:

```
define pnset foo stopped && { isactive(x) && (x > 0) }
```

This ensures that the evaluation takes place only on nodes that are stopped and on which **x** is active.

If an evaluation fails, the membership of the PN set remains what it was before you issued the **eval pnset** command.

You can use the **eval pnset** command in event actions; see Section 10.5.1.

Note the difference between *dynamic* and *variable* PN sets. The membership in both can change as a program executes. Dynamic PN sets are predefined sets like **stopped** and **interrupted**; Prism automatically updates their membership as the program executes. Variable PN sets are defined by the user, and the user must explicitly update their membership by issuing the **eval pnset** command.

Copyright © 1991–1994 Thinking Machines Corporation

10.3.4 Viewing the Contents of PN Sets

From the PN Sets Window

The easiest way to view the contents of PN sets is to use the **PN Sets** window.

By default, the window displays the current PN set (which starts out being the predefined PN set **`all`**), and the PN sets **`break`**, **`running`**, and **`error`**. When you create a new PN set via the **`define pnset`** command, that set is also displayed automatically.

To display a PN set, choose the **Show** selection from the **Options** menu in the **PN Sets** window This displays a list of PN sets; the predefined PN sets are at the top, followed by any user-defined set names. Click on a set name, and that set is displayed in the window.

To hide a PN set, choose the **Hide** selection from the **Options** menu. Once again, this displays the list of predefined and user-defined PN sets. Click on a set name to remove that set from the display.

Note that hiding a PN set doesn't otherwise affect its status; it still exists and can be used in commands.

Note also that there are choices **All Sets** and **all** in the **Show** and **Hide** submenus. The **All Sets** choice refers to all PN sets; the **all** choice refers to the predefined PN set **`all`**.

If you have too many PN sets to be shown in the display window of the **PN Sets** window, the navigator rectangle to the right of the **Cycle** arrows lets you pan through the PN sets. The white box in the rectangle shows the position of the display area relative to all the PN sets that are to be displayed:

You can either drag the box or click at a spot in the rectangle. The box moves to that spot, and the display window shows the PN sets in this area of the total display.

To display more PN sets at the same time, click on the **Zoom** up arrow to the right of the navigator rectangle. This reduces the size of the boxes representing the PN sets. Clicking on the **Zoom** down arrow increases the size of these boxes. By default, the boxes are at their highest zoom setting.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

From the Command Line

Use the **show pnset** command to print the contents of the PN set you specify. For example, the command

```
show pnset stopped
```

might produce this response:

```
The set contains the following PNs: 0:3.
```

The **show pnset** command is discussed further in Section 10.3.6.

The **show pnsets** command displays the contents and status of all PN sets.

```
(prism all) show pnsets
foo:
  definition = 0:31:2
  members = 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
  current pn = 0
break:
  definition = break
  members = nil
  current pn = (none)
done:
  definition = done
  members = 0:31
  current pn = 0
interrupted:
  definition = interrupted
  members = nil
  current pn = (none)
error:
  definition = error
  members = nil
  current pn = (none)
running:
  definition = running
  members = nil
  current pn = (none)
stopped:
  definition = stopped
  members = nil
  current pn = (none)
current:
  definition = 6, 9, 12
```


Copyright © 1991–1994 Thinking Machines Corporation

```
    members = 6,9,12
    current pn = 6
  cycle:
    definition = 6, 9, 12
    members = 6,9,12
    current pn = 6
  all:
    definition = all
    members = 0:31
    current pn = 12
```

10.3.5 Deleting PN Sets

You can delete named PN sets that you have defined. You cannot delete any predefined PN set except **`cycle`**; see Section 10.3.8. To delete a PN set:

From the PN Sets window: Choose the **Delete** selection from the **Options** menu. This displays a list of PN sets that you can delete. Click on the name of the PN set you want to delete. If it is currently displayed in the **PN Sets** window, it disappears.

From the command line: Issue the **`delete pnset`** command, using a PN-set qualifier to specify the name of a user-defined PN set. For example,

```
delete pnset xon
```

deletes the PN set named **`xon`**.

See Section 10.5.1 for a discussion of the effect of deleting a PN set on events that have been defined to affect the members of that set.

10.3.6 The Current PN Set

The command syntax described in Section 10.3.3 lets you apply a command to a specific PN set. If you don't use this syntax, the command is applied to the *current* PN set; **`current`** is a predefined PN set name in Prism. In addition, many graphical actions in Node Prism apply only to the members of the current set.

When a program is first loaded, the current PN set is the default PN set, **`all`**.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

You can change the current PN set via the **PN Sets** window or from the command line.

From the PN Sets window: There are several ways of changing the current PN set via the **PN Sets** window:

- If the set is displayed in the **PN Sets** window, simply double-click anywhere in its display (for example, on its name, or in the box beneath its name).
- Choose the **Set PN set** selection from the **Options** menu. This displays a list of PN sets. Click on the name of the set you want to be current.
- Edit the name of the PN set in the box below **Current Set** at the top right of the **PN Sets** window, then press **Return**.

When you change the current set, the new name appears in the **Current Set** box in the **PN Sets** window, and the current set shown at the top left of the PN sets area changes to reflect the contents of the new set.

From the command line: Issue the **`pnset`** command. For example,

```
pnset foo
```

changes the current PN set to **`foo`**.

You can also use the **`pnset`** command with the PN set-specification syntax described in Section 10.3.3. For example,

```
pnset 0:15:3
```

You cannot change the current PN set to one that has no members. If you try to do so, nothing happens in the **PN Sets** window, and you get a message like this one in the history region of the command window:

```
Cannot set current PN set to running -- it is empty.
```

Finding Out the Current PN Set

Node Prism provides many ways of finding out the current PN set:

- As described in the previous section, the name of the current PN set appears in the **Current Set** box at the top right of the **PN Sets** window.

Copyright © 1991–1994 Thinking Machines Corporation

- The name of the current PN set appears in the status region in Node Prism's main window.
- Issuing the **pnset** command without arguments displays the current set.
- The **(prism)** prompt on the command line and in commands-only Node Prism identifies the current PN set. For example, Prism's response to the **pnset** command in the previous section would look like this:

```
(prism all) pnset foo
(prism foo)
```

  NOTE: In giving examples of Node Prism commands, we use the **(prism)** prompt only when necessary to show the effect of a command.

To list the nodes in the current PN set, issue the **show pnset** command without arguments:

```
(prism foo) show pnset
PN set 'current' is defined as 'foo'.
The set contains the following PNs: 1,2.
```

The **PN Sets** window also displays the nodes in the current PN set.

The Current PN Set and Dynamic PN Sets

Section 10.3.2 described dynamic PN sets — predefined sets like **running**, **stopped**, and **interrupted**, whose contents Prism automatically updates during the execution of the program.

If you choose a dynamic PN set to be the current PN set, you create a *static* PN set that consists of the nodes that are members of the dynamic set at the time you issue the **pnset** command (or otherwise choose it to be the current set). To make this clear, the **(prism)** prompt changes to list the nodes that are members of this static set. For example, if nodes 0, 1, and 13 are the only nodes that are stopped, the **pnset** command has this effect:

```
(prism all) pnset stopped
(prism 0:1, 13)
```


Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

Output of the **show pnset** command is explicit under these circumstances:

```
(prism all) pnset stopped
(prism 0:1. 13) show pnset
The current set was created by evaluating the PN set
'stopped' once at the time when it became the current
set.
The set contains the following PNs: 0:1, 13.
```

Issuing the **pnset** command with no arguments displays the same information.

Note that the **(prism)** prompt can become quite long if there are many nodes in a current PN set derived from a dynamic PN set. You can limit the length of the prompt by issuing the **set** command with the **$prompt_length** variable, specifying the maximum number of characters to appear in the PN set part of the prompt. For example, this command shortens the prompt **long_pnset_name** to **long_pnset**:

```
(prism long_pnset_name) set $prompt_length=10
(prism long_pnset)
```

The Current PN Set and Variable PN Sets

Section 10.3.3 describes how to create variable PN sets — that is, user-defined PN sets whose membership can change in the course of program execution. You use the **eval pnset** command to update the membership of a variable PN set. If you make a variable PN set your current set, its membership is determined by the most recent **eval pnset** command you have executed for the set. If you have not executed an **eval pnset** command to update the set's membership, the membership continues to be what it was when you created the set.

10.3.7 The Current PN

Each PN set has a current PN. The current PN has a variety of uses in Node Prism:

- The source window displays the source code executing in the current PN of the current PN set.

Copyright © 1991–1994 Thinking Machines Corporation

- The **Where** graph centers around the call stack of the current PN set's current PN; see Section 10.5.2.
- The **Performance Data** window by default shows performance data for the current PN of the current PN set; see Section 10.7.
- The current PN determines the scope used in interpreting the names of variables. See Section 10.5.3.

By default, the current PN is the lowest-numbered PN in the set. You can change this as described below.

From the PN Sets window: Use one of these methods to change the current PN via the **PN Sets** window:

- Click on the cell representing the node in the displayed PN set. The cell turns gray (or, on color workstations, a darker shade of the color for the other PNs).
- To change the current PN in the current PN set, you can also edit the number in the box under **Node** at the top right of the window, then press **Return**.

From the command line: Issue the **pn** command to specify another current PN for the current PN set. For example,

```
(prism all) pn 2
The current PN is now 2.
```

Issue the **pn** command without any arguments to print the current PN of the current PN set.

When you change a current PN, by any of the methods described above, the PN set keeps this new current PN until you explicitly change it. That is, if you switch to a different current set, then switch back to the original set, the original set will still have the same current PN.

10.3.8 The Cycle PN Set

In debugging a message-passing program, you may often want to look in turn at each individual node within a PN set — for example, to see what the problem is for each node in the **error** PN set. The **cycle** PN set provides you with a convenient way of doing this.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

You create a **cycle** PN set out of an existing PN set. If the existing set is dynamic, the **cycle** set is statically fixed when you create it. You can then cycle through each PN in this set to examine it in turn.

By default, the **cycle** set is equivalent to the current set. You can set it to some other set via the **define pnset** command, as described in Section 10.3.3. For example,

```
(prism all) define pnset cycle foo
```

copies **foo** into the **cycle** set.

You can cycle through the nodes in the **cycle** set as follows:

From the PN Sets window: Use the **Cycle** arrows at the top left of the window to cycle through the members of the **cycle** set. Click on the right arrow to cycle up through the members of the set; click on the left arrow to cycle down through the members.

Clicking on a **Cycle** arrow does the following:

1. It advances the current PN in the **cycle** PN set to be the next member in the set.
2. It makes the current PN set consist of only this PN.

From the command line: Use the **cycle** command. This has the same effect as clicking on the right cycle arrow in the **PN Sets** window. For example, this Prism session defines a PN set, makes it the current set, and then cycles through its members:

```
(prism all) define pnset foo 0:3
(prism all) pnset foo
(prism foo) cycle
(prism 1) cycle
(prism 2) cycle
(prism 3) cycle
(prism 0)
```

Note that changing the **cycle** PN set erases any previous cycling information. For example, if you do the following:

1. Make **foo** the current set and cycle partway through it.
2. Make **bar** the current set.
3. Once again make **foo** the current set.

Copyright © 1991–1994 Thinking Machines Corporation

then you start at the beginning again when you cycle through the members of **foo**.

From the source-window popup menu: Choose **Cycle** from this menu to advance to the next member of the **cycle** PN set.

The Cycle Visualizer Window

Node Prism includes a **Cycle** window type for visualizing data. When you print a variable's value to the **Cycle** window, the value changes to that of the variable in the new node whenever you cycle through the members of the **cycle** PN set. For more information, see Section 10.6.

10.3.9 Using PN Sets in Commands

As we mention at the beginning of Section 10.3, some commands can take a PN set as a qualifier; they are listed at the end of this section. Put this qualifier after any arguments to the command, but before the optional **on** *window* syntax that specifies the window in which output is to be displayed. A command with a PN-set qualifier applies only to the nodes in the set. If you omit the qualifier, the command applies to the nodes in the current set.

Thus,

```
stop at 12 pnset error
```

sets a breakpoint at line 12 for the nodes in PN set **error**.

```
where pnset 0:10 on dedicated
```

displays the **Where** graph for nodes 0 through 10. See Section 10.5.2 for a description of the **Where** graph.

```
trace at 12 if x > 10
```

creates a trace event for the members of the current PN set.

Note that this last command applies only to the members of the current PN set. To apply it to all nodes, use the syntax

```
trace at 12 if x > 10 pnset all
```


Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

Many commands, of course, cannot logically take a PN-set qualifier. You get an error message if you try to issue one of these commands with a PN-set qualifier.

Here is the list of Prism commands that *can* take a PN-set qualifier:

address /
assign
call
catch
cont, contw
display
dump
ignore
interrupt
next, nexti,
pnstatus
print
return
step, stepi
stop, stopi
trace, tracei
wait
whatis
where

10.4 Executing a Program in Node Prism

You start execution of a program in Node Prism just as you do in Global Prism — by issuing the **run** command or choosing the **Run** or **Run (args)** selection from the **Execute** menu. See Section 3.5. You can also attach to an already-running program using the **attach** command (see Section 3.3). However, since a message-passing program has a more complicated structure that most programs, with both a host portion and a node portion, it may be useful to understand in more detail what is happening when you run such a program under Node Prism. This is explained in the sections below.

Copyright © 1991–1994 Thinking Machines Corporation

10.4.1 The Structure of a Message-Passing Program

A message-passing program contains a host portion and a node portion. A so-called "hostless" message-passing program still has a host portion, but the host portion consists solely of CMMD-supplied code.

If you only need to debug the node portion, the simplest thing to do is to use Node Prism, and start up your program with the **`run`** command.

If you need to debug both the host portion and the node portion, then you'll need two copies of Prism. One way of doing this, as described in Section 10.2.2, is to load the program into Global Prism, then click on **PN Debug** in the **Utilities** menu to start up Node Prism. A disadvantage of this method is that you get the commands-only version of Node Prism. A more flexible method is to start up both Prisms (either the graphical or the commands-only version), one with the **`-node`** option and one without. Load your program into both of them. Start up your program using **`run`** from either of the Prisms, and attach to it using **`attach`** from the other Prism (see Section 10.4.3). What you end up with is shown in Figure 41.

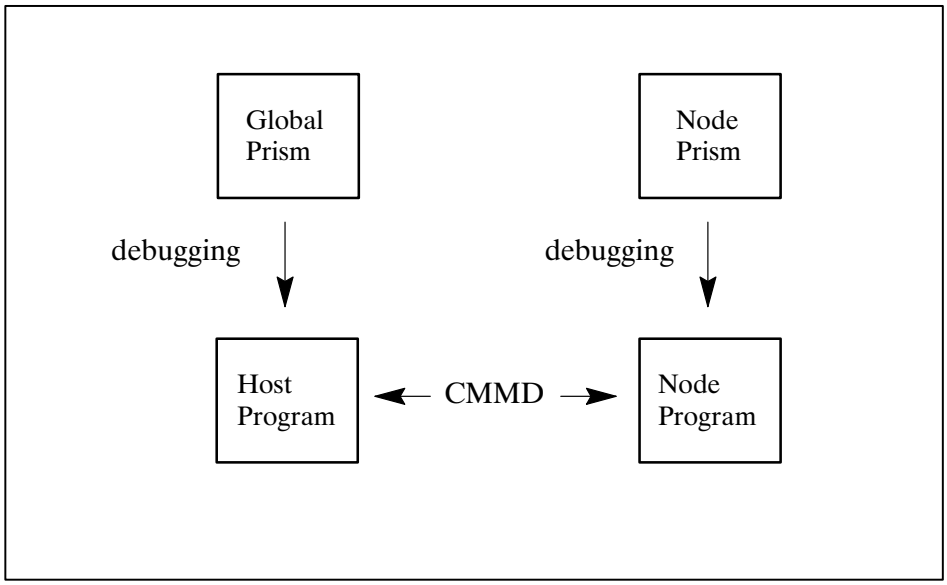

Figure 41. Using two Prisms to debug a message-passing program.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

10.4.2 Running

The **run** command starts execution of your program. Here is a commands-only Node Prism session that loads a program, sets a breakpoint, and runs:

```
% prism -C -node
(prism all) load chess
reading symbolic information from chess...
(prism all) stop in main
(1) stop in main pnset all
(prism all) run
PNs 0:3: attached to process 12577
PNs 0:3: stopped in procedure "main" at address 0x2c7c
```

Internally, **run** is doing the following:

- starting up the host portion of your program
- temporarily stopping the host program at the **cmmd_debug** label
- attaching Node Prism to the node portion of your program
- allowing the host program to run freely
- continuing the node portion of the program on all nodes, so they are in the **running** state

For a CMMD program, **cmmd_debug** is the correct place to pause execution of the host program. However, for other kinds of programs, it may not be correct. For example, if you want to use Node Prism to inspect what the nodes are doing in a global CM Fortran program, then the right place to pause execution is the procedure **MAIN**. Use the environment variable **HOST_PAUSE_POINT** to specify this different pause point. For example,

```
(prism all) run
Can't start up host portion of .../cmf/arrays.x
 - couldn't find 'cmmd_debug' global.
Start up the program in a separate window and use
'attach'
(prism all) setenv HOST_PAUSE_POINT MAIN
(prism all) run
PNs 0:3: attached to process 12601
```

If you run a program using the **run** command, your process will be killed if any of these conditions are met:

- You quit Prism.

Copyright © 1991–1994 Thinking Machines Corporation

- You rerun the program.
- You run another program.
- You attach to another program.

10.4.3 Attaching

You can attach Node Prism to an already-running CM process by issuing the **attach** command. You give it the process ID of the running process, which you can find out by issuing the CMOST command **cmps**. Note that you must previously have loaded the correct executable program (the one matching the running process), or else you will obtain confusing results.

The **attach** command leaves all PNs in the **interrupted** state.

An example:

```
(prism all) sh cmps
4 PN System, 26524K mem. free, 6628K VU mem. free, 2 procs, TS-9/9/93-11:51 (CMOST 7.3
Beta 1) Daemon up:  1:20

USER      PID   CMPID  TIME   TEXT    ILH     ILS     IGS   IGH   VUS  VUH   COMMAND
title     12591 2      9:46  1016K  1164K     48K      0K   0K     0K   0K prism_cmmd
title    *12649 1      0:51   456K  1352K     48K      0K   0K     0K   0K
chess
(prism all) load chess
reading symbolic information from /tmp_mnt/users/cmsg7/title/mychess/cmmd/chess.
.
(prism all) attach 12649
PNs 0:3: attached to process 12649
PN 0: interrupt in procedure "cmaml_internal_dispatcher" at address 0x2adb8
PN 1: interrupt in procedure "in_check" at "chess.c":784
PN 2: interrupt in procedure "generate_moves" at "chess.c":882
PN 3: interrupt in procedure "make_move" at "chess.c":1167
```

If you attach to a program under Node Prism, your process will be automatically detached from Node Prism if you quit, run another program, or attach to another program.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

10.4.4 Detaching

The **detach** command can be used to set the debugged process free. You can detach from your process if you started it up with **run**, and, of course, you can also detach if you previously attached.

Node Prism lets you detach only when all your PNs are stopped (to make sure you remove all breakpoints before detaching). The detach operation itself sets them all running again, outside control of the debugger.

10.4.5 Quitting

Issuing the **quit** command terminates the debugging session. As mentioned above, before quitting, Node Prism will kill your debugged process if it was started with **run**, or detach from it if you previously attached.

10.4.6 Using Prism with Global/Local CM Fortran Programs

You can use Prism Version 2.0 to debug both the global and the local part of a global/local CM Fortran program. However, you will need two copies of Prism (one to debug the global part and one to debug the local part).

To get two graphical Prisms started on your program, you can follow this procedure. (Assume your program is called **a.out**.)

1. In one window, issue the command:

   ```
   % prism a.out
   ```

   Prism starts up.

2. Set a breakpoint in **a.out** and issue a **run** command to get your program going.

3. Note the program's process ID, which is listed in a Prism message.

4. Then issue the **prism -node** command, specifying the name of the program and its process ID:

   ```
   % prism -node a.out pid
   ```


Copyright © 1991–1994 Thinking Machines Corporation

This starts the second copy of Prism. You may have to use the **file** command to manually set Prism to the top-level local file. All nodes will come up in interrupted state. You can set breakpoints in your local code if you wish; then you should issue a **cont** command to get all nodes running again.

At this point you have two copies of Prism running. You can use Global Prism to perform any debugging operations on the global part of your program, and Node Prism to perform any debugging operations on the local part of your program.

Note, however, that if an array does not neatly match the size of the machine, so that some PNs have less data than others (or possibly no data at all) then you must debug on a node-by-node basis; printing data with the PN set **all** in Node Prism will give incomplete, incorrect, or meaningless results.

Example

Let's assume that the global part of your program looks like this:

```
program main
real, array (16, 16) :: a
   ...
call local_routine(a)
   ...
end
```

And the local part of your program looks like this:

```
subroutine local_routine(a)
real, array (:,:) :: a
   ...
end
```

If you print **a** in Global Prism, you will get a 16x16 array. If you use Node Prism to stop in **local_routine**, and you print **a** there, you will see a smaller array in each PN, representing that PN's slice of **a**. (See Section 10.6 for information on printing in Node Prism.) For instance, if your partition has 4 PNs, you will see a 4x16 array in each PN.

NOTE: In order to do debugging operations on CM arrays from Global Prism (the global **print a** in the above example), all PNs must be running from the point of view of Node Prism — that is, they cannot be interrupted

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

or stopped at Node Prism breakpoints. If any of the PNs are stopped, the global print will hang until they are continued.

Finally, note that performance data is not currently available for the local portion of a global/local program.

10.4.7 Stepping and Continuing through a Program

In Node Prism, menu actions such as **Step** and **Next** apply to the nodes of the current PN set.

Global Prism (like other debuggers) waits for a **`step`**, **`next`**, or **`cont`** command to finish executing before letting you issue most other commands. This behavior is unnecessarily restrictive in Node Prism, however; therefore, if one node or set of nodes is executing code, you can still issue commands that affect other nodes. For example, you can switch to a different PN set and start or stop execution in its nodes.

10.4.8 Interrupting and Waiting for Nodes

It is useful in debugging message-passing programs to wait for a specific node or nodes to stop executing, or to be able to interrupt execution on individual nodes. Node Prism therefore provides the commands **`interrupt`** and **`wait`**.

Use the **`interrupt`** command to forcibly interrupt execution on a specified node or nodes. For example,

```
interrupt pnset 0
```

interrupts execution on PN 0.

```
interrupt pnset running
```

interrupts all running PNs.

Using the **`interrupt`** command resets the predefined PN set **`interrupted`** so that it includes the newly interrupted PNs. Nodes leave this PN set when they continue execution.

Copyright © 1991–1994 Thinking Machines Corporation

In Node Prism, the **Interrupt** selection from the **Execute** menu interrupts nodes in the current PN set that are running.

Use the **wait** command to wait for a specified node or nodes to stop execution. A node is considered to have stopped if it has entered the **done**, **break**, **interrupted**, or **error** state.

There are two versions of the **wait** command:

- Use the syntax **wait** or **wait every** to wait for every member of the specified PN set to stop. If no PN set is specified, the command applies to the current PN set. Thus,

  ```
  (prism notx) wait every
  ```

  waits for every node in the PN set **notx** to stop. The current PN will be whatever it would normally be; see Section 10.3.6. This is the default behavior of the **wait** command.

- Use the syntax **wait any** to wait for any member of the specified PN set to stop. If no PN set is specified, the command applies to the current PN set. When the first node stops, it becomes the current PN of this PN set. Thus,

  ```
  wait any pnset foo
  ```

  waits for the first node in PN set **foo** to stop.

There are corresponding **Wait Any** and **Wait Every** selections in Node Prism's **Execute** menu. They apply to the nodes of the current set.

You can end the wait by

- typing **Ctrl-c**; this does not affect nodes that are running
- choosing the **Interrupt** selection from the **Execute** menu; this stops nodes that are running, as well as ending the wait

Note that, if you prefer that **step** and **next** commands wait for nodes to finish executing before letting you issue other commands, you can issue them along with the **wait** command. For example,

```
step; wait
```

This says: Execute the next line, then wait for all nodes in the current PN set to finish execution.

Copyright © 1991–1994 Thinking Machines Corporation

If you use this command sequence frequently, you can provide an alias for it via the **alias** command. Prism provides the default alias **contw** for the commands:

```
cont; wait
```

10.4.9 The Execution Pointer

In Global Prism, the **>** symbol in the line-number region points to the next line to be executed; see Section 2.6.2. In a message-passing program, there can be multiple execution points within the program. Node Prism marks all the execution points for the PNs in the current set by a **>** in the line-number region (or a ***** if the current source position is the same as the current execution point). Shift-click on this symbol to display a popup window that shows the node(s) for which the symbol is the execution pointer.

10.4.10 Finding Out Execution Status

Issue the **pnstatus** command to find out the execution status of nodes. Without a PN-set qualifier, it displays the execution status of the members of the current set. For example,

```
(prism all) pnstatus
PN 0: running
PN 1: stopped in procedure ”pawn_moves” at ”chess.c”:49
PN 2: interrupted in procedure ”construct_move” at
”chess.c”:1187
PN 3: interrupted in procedure ”rook_check” at
”chess.c”:746
```

Use a PN-set qualifier to find out the execution status of the members of the specified PN set.

10.4.11 Executing a Program in Commands-Only Node Prism

When you issue the **run** command to execute a program in the commands-only version of Node Prism, the program starts up in the background. If the program needs to read terminal input, you must then issue the **fg** command

Copyright © 1991–1994 Thinking Machines Corporation

at the **(prism)** prompt to run the program in the foreground. You cannot execute Prism commands while the program is executing in the foreground. To have the program run in the background and regain the **(prism)** prompt, type **Ctrl-z**.

10.5 Debugging in Node Prism

Debugging a message-passing program can be considerably more complex than debugging a data parallel program, since you are in effect debugging multiple individual programs concurrently, each on a separate node. Node Prism's concept of PN sets lets you focus your debugging efforts on the nodes that are of particular interest. This section describes the following areas in which debugging in Node Prism, with its PN sets, is different from debugging in Global Prism (described in Chapter 4):

- events; see Section 10.5.1
- the **Where** graph; see Section 10.5.2
- scope; see Section 10.5.3
- examining vector-unit registers; see Section 10.5.4
- examine node core files; see Section 10.5.5

10.5.1 Events in Node Prism

Events in Node Prism can take a PN-set qualifier. You can specify this in an event field in Node Prism's event table, as shown in Figure 42.

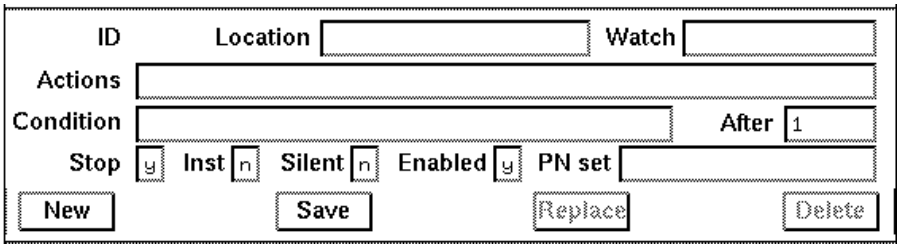

Figure 42. The PN set field in Node Prism's event table.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

If you don't supply a PN-set qualifier, the event applies to the current PN set. If you create the event before changing the current set, the event applies to the default set, which is **all**.

Thus,

```
stop in receive pnset notx
```

sets a breakpoint in the **receive** routine for the PNs in the set **notx**. Each PN in the set stops when it reaches this routine. It is possible, of course, that some nodes may never reach this routine. This becomes an issue when you include actions in an event; see below.

If all the nodes in the PN set have stopped, you can continue them by issuing a command like

```
cont pnset notx
```

Here is another example:

```
stop if x > 10
```

This command stops execution for any node in the current PN set if the node's value for the scalar variable **x** is greater than 10.

Prism evaluates the expression in the condition locally — that is, separately for each node. Similarly, if **a** and **b** are arrays or parallel variables,

```
stop if sum(a) > sum(b)
```

stops execution for a node in the current set if the sum of the values of **a** in that node is greater than the sum of the values of **b**.

All nodes that are stopped at breakpoints are members of the predefined PN set **break**.

Events and Dynamic PN Sets

If you are using a dynamic PN set as a qualifier for an event, its membership is evaluated when you issue the command defining the event. Thus, the command

```
stop at 10 pnset interrupted
```


Copyright © 1991–1994 Thinking Machines Corporation

creates a breakpoint only in the nodes that are interrupted at the time the command is issued. If no nodes are currently interrupted, you receive an error message.

One result of this is that you cannot define events that involve dynamic PN sets before the program starts execution.

Events and Variable PN Sets

If you are using a user-defined variable PN set as a qualifier, its membership is determined by the most recent **`eval pnset`** command you issued for that PN set.

As is the case with dynamic PN sets, you cannot define events that involve variable PN sets before the program starts execution.

Actions in Events

Events in Node Prism can take action clauses, just as they can in Global Prism. For example,

```
stop at 10 {print x} pnset foo
```

prints **`x`** for the PN set **`foo`** when the members of **`foo`** are stopped at line 10.

IMPORTANT: Associating an action with an event forces a global synchronization at the breakpoint or tracepoint. In the example above, every node in PN set **`foo`** must stop at line 10 before **`x`** can be printed. If a member does not stop at line 10, the action never takes place. In a trace event, all nodes in the PN set must stop at the specified place and synchronize; the action then takes place, and the nodes automatically continue execution.

You can include an **`eval pnset`** command as an event action. For example,

```
stop in send {eval pnset sending}
```

evaluates the PN set **`sending`** when all the members of the current PN set are stopped in **`send`**. You receive error messages if it is impossible to evaluate membership in a node (for example, because a variable in the set definition is not active).

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

Note these limitations in using event actions in Node Prism:

- You cannot include the following commands that manipulate PN sets:

  define pnset
  delete pnset
  pn
  pnset

- You cannot include a PN-set qualifier in the action. The command in the action clause takes its PN set from the PN set of the event.

- As in Global Prism, you cannot include commands that affect program execution, specifically

  cont and **contw**
  run
  step and **stepi**
  next and **nexti**
  wait

- As in Global Prism, you cannot include the **load**, **reload**, **return**, and **core** commands.

Displaying Events by Node

Issue the **show events** command with a PN number as an argument to display all events associated with that node. For example,

```
(prism all) show events 0
(1) stop at 10 pnset 0
(3) stop at 575 {print x] pnset all
(7) trace y pnset bar
```

Issuing **show events** with no arguments has its standard behavior; that is, it prints out all events.

Events and Deleted PN Sets

If you create an event that applies to a particular PN set, and subsequently delete the PN set, the event continues to exist. Its printed representation, however, is changed so that it shows the nodes that were members of the PN set at the time you deleted the set.

Copyright © 1991–1994 Thinking Machines Corporation

Using the Line-Number Region

Section 2.6.2 describes how the line-number region displays breakpoints and tracepoints in Global Prism. Node Prism provides a variation of this feature:

- It displays a **B** next to a line number if all nodes in the current PN set have a breakpoint set at that line.
- It displays a **b** if some but not all of the nodes in the current PN set have a breakpoint set at that line.
- It displays a **T** if all nodes in the current PN set have a tracepoint set at that line.
- It displays a **t** if some but not all of the nodes in the current PN set have a tracepoint set at that line.

If there is a mixture of breakpoints and tracepoints set on the line, Prism uses the **B**-**b**-**T**-**t** sequence to determines what letter to display. For example, if a line has a breakpoint set in one node and a tracepoint set in all nodes, Prism displays a **b**.

As in Global Prism, you can shift-click on the letter in the line-number region to display the complete event (or events) associated with it.

10.5.2 The Where Graph

In Global Prism, choosing **Where** from the **Debug** menu displays the call stack for the program; see Section 4.5. A message-passing program, however, can have multiple call stacks, one for each node that is executing the program. To show the relationships among these call stacks, Node Prism provides a *Where graph*; this window displays a snapshot of the dynamic call graph of the program. Information is displayed for all nodes that are not running.

To display the **Where** graph:

From the menu bar: Choose **Where** from the **Debug** menu.

From the command line: Issue the command

```
where on dedicated
```


Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

A window like the one shown in Figure 43 is displayed.

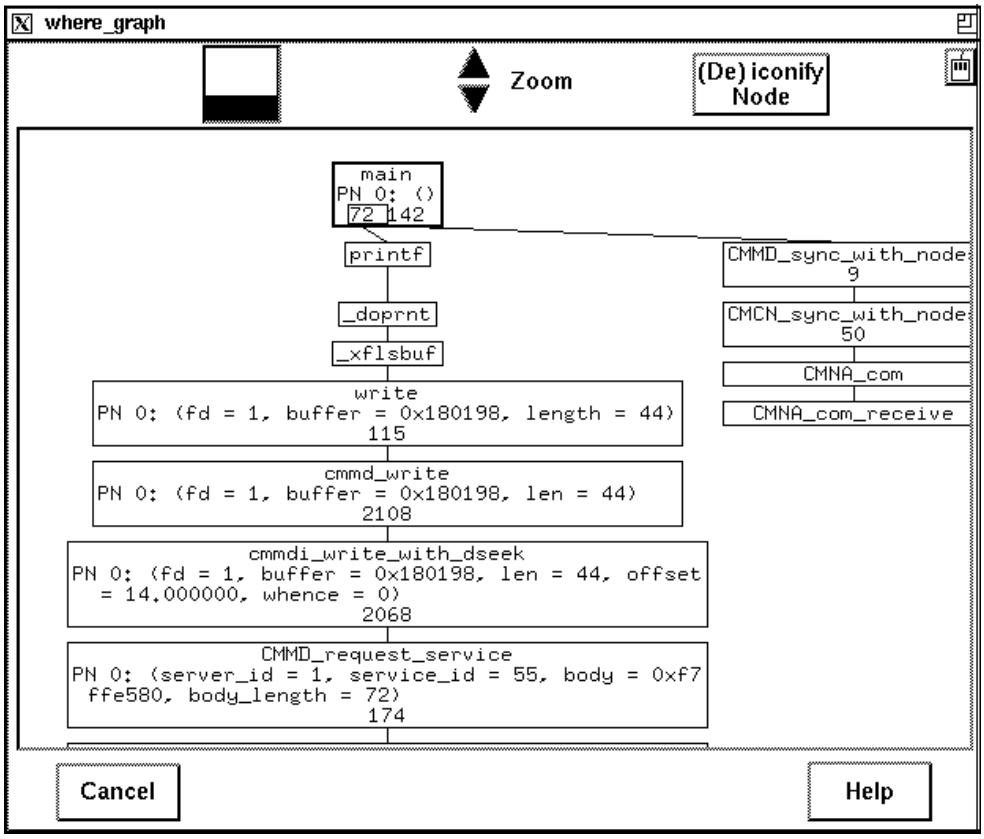

Figure 43. A Where graph.

The **Where** graph centers on the current PN of the current PN set — that is, the nodes related to it are lined up in a single column. In Figure 43, PN 0 is the current node. If you change the current PN, the **Where** graph rearranges itself. The default zoom level of the **Where** graph shows the arguments for the current PN.

At the bottom of each box are line numbers indicating where nodes branch. Thus, in Figure 43, one or more nodes call **`printf`** at line 72 of **`main`**, and one or more nodes call **`CMMD_sync_with_nodes`** at line 142.

Shift-click in each function's box to display a popup window showing the numbers of the nodes with this function in their call stack, along with their arguments.

Copyright © 1991–1994 Thinking Machines Corporation

Panning and Zooming in the Where Graph

As Figure 43 shows, the **Where** graph can get quite large, so Node Prism provides methods for panning through it and zooming in and out.

The white box in the navigator rectangle at the top of the window shows the position of the display area relative to the entire **Where** graph. You can either drag the box or click at a spot in the navigator. The box moves to that spot, and the window shows the **Where** graph in this area of the total display.

To display more of the **Where** graph at the same time, click on the **Zoom** down arrow to the right of the navigator. This reduces the size of the boxes representing the functions and removes information. Figure 44 shows the **Where** graph of Figure 43, zoomed out one level. Note that the information about the current PN's arguments is gone.

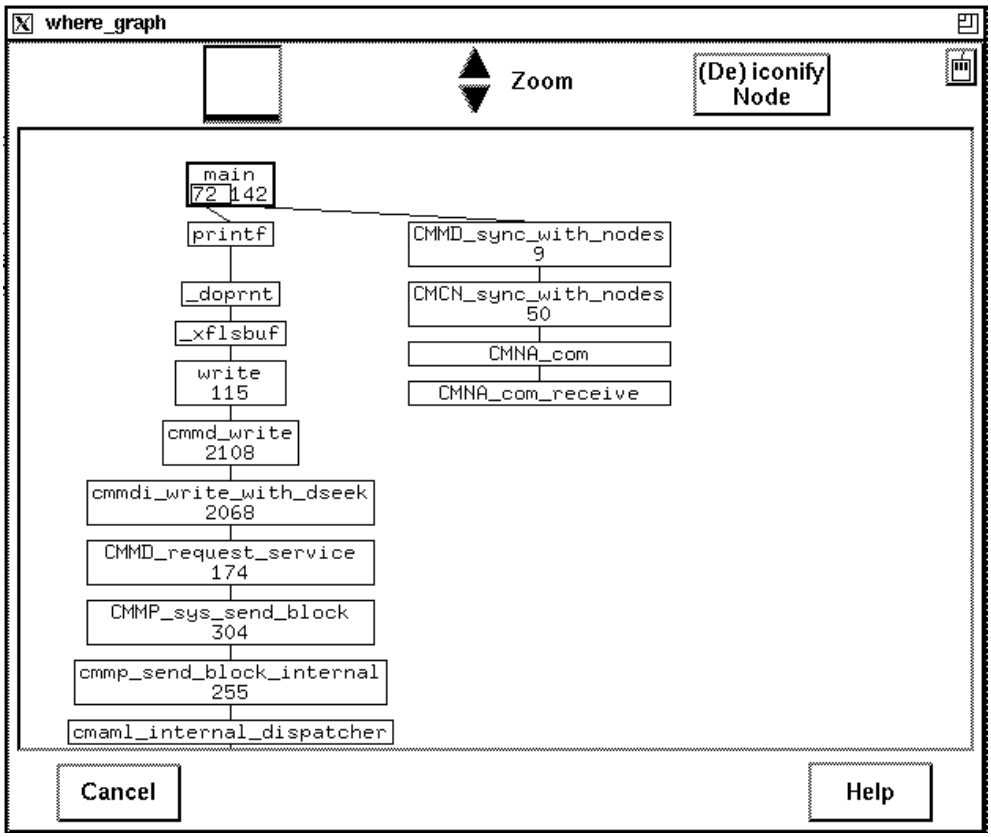

Figure 44. A Where graph, zoomed out one level.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

Zooming out one more level removes the line numbers, and one more level after that removes the function names, leaving only boxes connected by lines, as shown in Figure 45. You can still shift-click on a box to display information about it.

Clicking on the **Zoom** up arrow increases the size of the function boxes and includes more information in them. Figure 45 shows the **Where** graph of Figure 43, zoomed in. In this case, the **Where** graph shows, for each function, the nodes that have that function in their call stack. As in the **PN Sets** window, the nodes are represented as bitmaps of cells, numbered starting at the upper left, increasing from left to right and then jumping to the next row.

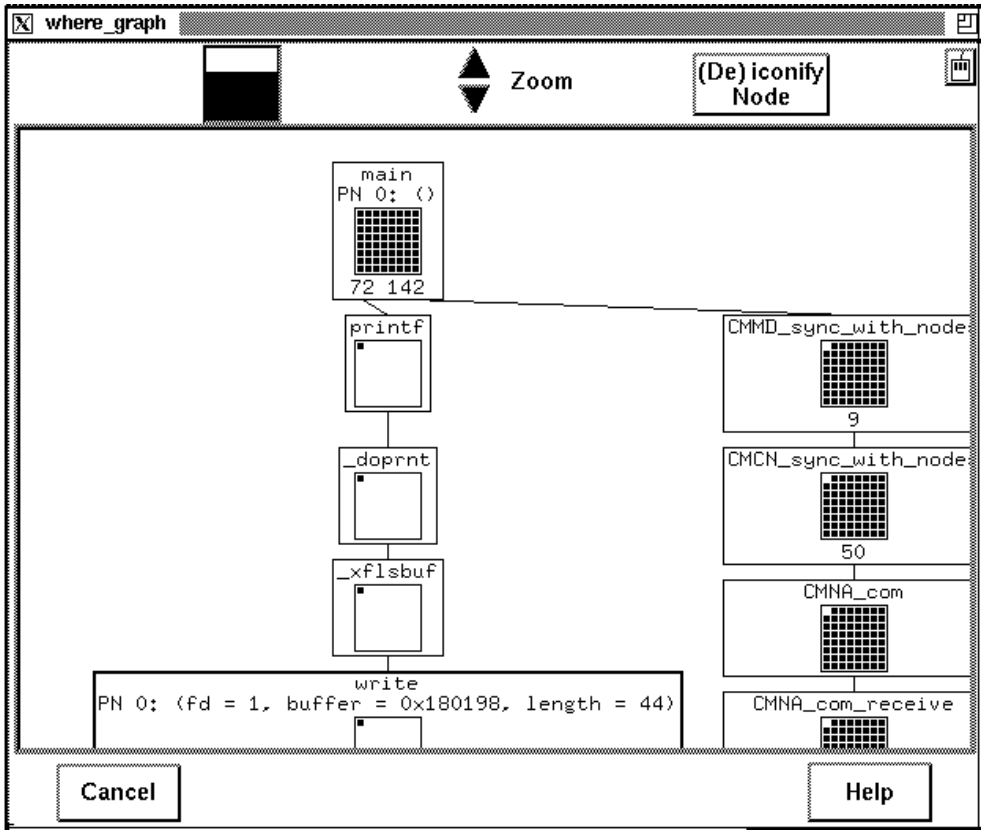

Figure 45. A Where graph, zoomed in.

Zooming in another level shows all arguments for all nodes.

Copyright © 1991–1994 Thinking Machines Corporation

Shrinking Selected Portions of the Where Graph

You can shrink selected portions of the **Where** graph. This is useful if you want to see the overall structure of the graph, but in addition want to focus on certain functions.

Middle-click on a function to iconify it and all of its children. Middle-click on an iconified function to re-expand it and its children to the current zoom level.

Alternatively, you can click on the **(De)iconify Node** button next to the **Zoom** arrows at the top of the **Where** graph. This changes the mouse pointer to a target. You can then left-click on a function to iconify it and its children. If it is already iconified, left-clicking on it will re-expand it and its children. To cancel the operation, left-click anywhere outside of the boxes surrounding the functions.

Moving through the Where Graph

When you first display the **Where** graph, the `main` function is highlighted. You can left-click on a function to highlight it. Or, you can move through the **Where** graph via the keyboard:

- Use the up arrow key to move to the parent of the highlighted function.
- If line numbers are visible in the highlighted function, by default the leftmost number is selected by having a box drawn around it. Use the left and right arrows to select other line numbers in the function. You can then use the down arrow key to highlight the function called at the selected line.

Making a Function the Current PN Set

Pressing the spacebar while in the **Where** graph does the following:

- It changes the current function to be the function that is highlighted in the **Where** graph.
- This function is displayed in the source window.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

- It creates a new current PN set, with the same name as the function, and containing the nodes with this function in their call stack. The current PN of this current set is the lowest-numbered PN in the set.

Issuing the where Command in Node Prism

Issuing the **where** command by default displays (in the history region) the call stack consecutively for each node in the current set (or in the PN set you specify via the **pnset** qualifier).

Issuing the command

```
where on dedicated
```

displays the **Where** graph, as described above.

Issuing the command

```
where on snapshot
```

puts the history-region output into a window; it does not create a **Where** graph.

10.5.3 Scope in Node Prism

See Section 4.5.2 for a discussion of scope in Global Prism.

In Node Prism, the scope of the current PN determines the scope for resolving the names of variables. See Section 10.3.7 for a discussion of the current PN.

If a command applies to a PN set other than the current set, Prism uses the scope of that set's current PN.

It is possible that other members of the PN set will have different scopes from that of the current PN, or that its scope level does not even exist on these nodes. In these cases, you receive an error message when you try to issue a command (for example, **print** or **display**) that requires a consistent scope. To solve the problem, you can do the following:

- Restrict your PN set so that it contains only members with the same scope.

Copyright © 1991–1994 Thinking Machines Corporation

- If the current PN's scope level does not exist on other nodes in the set, you can use the **up** command to move up its call stack to a point where it has a scope level that does exist on the other nodes.
- If different nodes in the set have different scopes, you can issue the **up** and **down** commands as needed to ensure that they all have the same scope.

Commands such as **pnset** and **pn** that affect scope print the current function when you issue them.

10.5.4 Examining the Contents of Vector-Unit Registers

Node Prism supports printing of and assigning to vector-unit registers. For complete information on these registers, see the *VU Programmer's Handbook*.

The Data Registers

For each PN, there are 4 VUs. Each VU contains a bank of 128 words of vector registers. For single-precision operations, these form 16 registers of eight words each. The registers are identified by the names **V0**, **V1**, ... **V15**. For double-precision operations, the vector registers are doubled up to form eight registers of eight doublewords each. These are named **V0**, **V2**, ... **V14**.

The data type of the register is implicit in the assembly instruction being performed. The possible data types are

- **i** – 32-bit **int**
- **di** – 64-bit **int**
- **u** – 32-bit **unsigned int**
- **du** – 64-bit **unsigned int**
- **f** – 32-bit **float**
- **df** – 64-bit **float**

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

For example, in DPEAC the following instruction adds the eight floating values in vector register **V0** to the eight floating values in vector register **V1**, leaving the result in vector register **V2**:

```
fadd V0,V1,V2
```

The following instruction adds the eight double-precision values in the register-pair {**V0**,**V1**} to the eight double-precision values in the register pair {**V2**,**V3**}, leaving the result in {**V4**,**V5**}:

```
dfadd V0,V2,V4
```

In Node Prism, each of the vector-register built-ins is a 4x8 array. The first dimension is for the four VUs, and the second dimension is for the eight elements in each vector register. If the PN set for which the data is being displayed contains more than one PN, Prism does an additional level of aggregation, with the nodes becoming the *first* dimension, the four VUs in the second dimension, and the eight elemnts in the third dimension.

Node Prism appends the data type to the name of the register, using the six type indicators above, and prepends a dollar sign, to avoid conflicts with variables in the user program.

Thus, Node Prism provides these built-in symbols:

$v0_i, **$v1_i**, ... **$v15_i**
: These are the 16 vector registers, each one being interpreted as a 4x8 array of integers.

$v0_u, **$v1_u**, ... **$v15_u**
: The 16 vector registers, each one being interpreted as a 4x8 array of unsigned integers.

$v0_di, **$v2_di**, ... **$v14_di**
: The 8 vector-register pairs, each one being interpreted as a 4x8 array of 64-bit integers.

$v0_du, **$v2_du**, ... **$v14_du**
: The 8 vector-register pairs, each one being interpreted as a 4x8 array of unsigned 64-bit integers.

$v0_f, **$v1_f**, ... **$v15_f**
: The 16 vector registers, each one being interpreted as a 4x8 array of **float**s.

Copyright © 1991–1994 Thinking Machines Corporation

$v0_df, **$v2_df**, ... **$v14_df**
: The 8 vector-register pairs, each one being interpreted as a 4x8 array of **double**s.

You can use all of these within Node Prism expressions in the usual way. For example, this command prints the contents of vector register 0 as an array of integers:

```
(prism 0) print $v0_i
$v0_i =
(0,0:7)   0  0  1  1  2  0  3  3
(1,0:7)   0  0  0  0  0  0  0  0
(2,0:7)   0  0  0  0  0  0  0  0
(3,0:7)   0  0  0  0  0  0  0  0
```

This command adds 1 to the integer values in (0,0) through (0,7) of vector register 0, and prints the result:

```
(prism 0) print $v0_i(0,:) + 1
$v0_i(0,:)+1 =
(0:7) 1  1  2  2  3  1  4
```

This command prints the integer value in (0,7) of vector register 0:

```
(prism 0) print $v0_i(0,7)
$v0_i(0,7) = 3
```

The Control Registers

Node Prism also provides a set of built-in symbols corresponding to the names of the VU control registers. These symbols in general represent 4-element arrays, one element for each VU. If more than one VU shares the same control register, Prism simply repeats the register's value in the corresponding elements of the array.

Here are Node Prism's names for the control registers:

```
$alu_mode
$bad_address_high
$bad_address_low
$bad_instruction_high
$bad_instruction_low
$current_element
$heap_limits
$instruction_ext
```


Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

```
$instruction_ext_enb
$interrupt_cause_green
$interrupt_cause
$interrupt_cause_green_stored
$interrupt_cause_stored
$interrupt_enable_green
$interrupt_enable
$memory_access_mode
$stack_limits
$status_enable
$status
$stride_rs1
$stride_memory
$vector_length
$vector_mask
$vector_mask_buffer
$vector_mask_mode
$vector_mask_direction
```

Examples:

```
(prism 0) print $vector_length
$vector_length =
(0:3)  8   8   8   8
(prism 0) print $stride_rs1
$stride_rs1 =
(0:3)  1   1   1   1
```

10.5.5 Examining Node Core Files

You can use Node Prism, with certain limitations, to examine node core files.

Description of Node Core Files and Associated Files

Certain errors can cause cores for some of the nodes to be dumped and an errors file to be generated. For example,

- a segmentation fault on the nodes
- a bus error on the nodes
- division by 0 on the nodes

Copyright © 1991–1994 Thinking Machines Corporation

These in turn can be caused by programming errors. For example, here are some C* errors that can cause core to be dumped:

- dereferencing a bad parallel pointer
- using an invalid send address
- doing out-of-bounds array referencing of parallel variables
- using an invalid parallel coordinate

Problems of this sort are generally signaled by an obscure CMOST message — for example,

```
CMOST: yellow interrupt
```

The errors file is called **`CMTSD_errors.`***pid*, where *pid* is the process ID of the process you were running; it is located in the directory from which you executed your program. The file is generated by the timesharing daemon when a user program crashes; it contains a list of the status of each node (and of the partition manager, if an error was detected there). The file tells you which nodes crashed, and give you some information about the crash, such as what memory address the node was trying to reference and whether it died because of a segmentation fault.

If your program was using the CM-5's vector units, additional error files are created in the directory from which you executed the program. Information on vector-unit registers is contained in the files **`CMTSD_dp.pn`***X*`.`*pid*, where *X* is the node identifier and *pid* is the process ID. These files contain an ASCII dump of the contents of every register on the vector units, for every node that has a different error. Similarly, vector-unit memory is captured in the files **`CMTSD_stack.pn`***X*`.`*pid* and **`CMTSD_heap.pn`***X*`.`*pid*.

The core files are named **`CMTSD_core_pn`***X*`.`*pid*, where *X* is the node identifier and *pid* is the process ID; like the errors files, they are created in the directory from which you executed your program. You may also see a regular core file for the partition-manager executable program.

To avoid wasting space, only unique cores are dumped. That is, if several nodes have the same error, only the core for the first node with that type of error is dumped. Also, the first node with no error (if there is such a node) will dump core.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

How to Look at a Node Core File

You can examine the node core files described above with Node Prism. To do this, specify the core filename on the command line, using this format:

```
% prism -node executable_name CMTSD_core_pnX.pid
```

where **CMTSD_core_pn***X*.*pid* is the core file for node *X*. When Prism comes up, it lets you look at the state of node *X*, and you can issue commands like **where** and **print** to inspect the state of your process on the node at the time the core dump was taken. But note these restrictions:

- You cannot switch PN sets.
- If node *Y* also dumped core, so that you have a core file **CMTSD_core_pn***Y*.*pid*, you must run Prism again on the *Y* core file to inspect the state of node *Y*.
- You cannot issue any execution commands (for example, **run**, **cont**, or **step**).
- You cannot change the values of variables via the **assign** command.
- You cannot get information about vector-unit registers and memory. As discussed above, this information is contained in the files **CMTSD_dp.pn***X*.*pid*, **CMTSD_stack.pn***X*.*pid*, and **CMTSD_heap.pn***X*.*pid*; these files are not accessible via Node Prism.

Also, note that you cannot use the **core** command to examine a core file once you have started Node Prism.

An Example

Suppose our program crashes and dumps core. It gets a bus error on node 0, a divide-by-zero on node 1, and no errors on the other nodes. We wind up with three core files, one for each error node and one example of the non-error nodes. For example,

```
CMTSD_core.pn0.18591
CMTSD_core.pn1.18591
CMTSD_core.pn2.18591
```


Copyright © 1991–1994 Thinking Machines Corporation

Let's run Node Prism in succession on each of the three core files, and do a **where** on each one to see where they were at the time the program crashed.

For node 0, Prism gives us the error that caused the core dump and shows where the error occurred. A further **print p** command tells us the value of **p**, and hence why the bus error occurred:

```
% prism -C -node myprog CMTSD_core.pn0.18591
Welcome to Prism, Version 2.0-final
Type 'help' for help.
reading symbolic information from myprog...
reading core file CMTSD_core.pn0.18591...
signal BUS (bus error) in procedure "main" at
"main.c":49
49        *p = *p;
(prism) where
main(), line 49 in "main.c"
(prism) print p
main`main`p = 0xffffffff
```

For node 1, Prism also gives us the error that caused the core dump and shows where the error occurred. A further **print k** command tells us the value of **k** (and hence why the statement in question gave an FPE error):

```
% prism -C -node myprog CMTSD_core.pn1.18591
Welcome to Prism, Version 2.0-final
Type 'help' for help.
reading symbolic information from myprog...
reading core file CMTSD_core.pn1.18591...
signal FPE (floating point exception) in procedure
".div" at address 0x802fc
Current function is main
52        k = 1/k;
(prism) where
.div() at 0x802fc
main(), line 52 in "main.c"
(prism) print k
main`main`k = 0
```


Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

For node 2, Prism tells us that no error occurred. The program location is simply where node 2 happened to be at the time the application was terminated:

```
% prism -C -node myprog CMTSD_core.pn2.18591
Welcome to Prism, Version 2.0-final
Type 'help' for help.
reading symbolic information from myprog...
reading core file CMTSD_core.pn2.18591...
signal (null) (no signal) in procedure "main" at
"main.c":80
80        for (j = 0; j <= MAX_DEPTH; j++)
(prism) where
main(), line 80 in "main.c"
```

10.6 Visualizing Data in Node Prism

See Chapter 6 for general information on visualizing data in Prism.

When you print or display an object in Node Prism, the data is shown for all nodes in the PN set you specify (in the current PN set, if you do not include a PN-set qualifier). Choosing the **Print** or **Display** selection from the **Debug** menu prints or displays data for nodes in the current PN set.

If there is only one node in the PN set, the visualizer that is displayed is no different from the visualizer you would see in Global Prism.

If there is more the one node in the PN set, Prism adds a dimension to the visualizer. The extra dimension represents the nodes in the set. For example, if the variable is scalar, Prism displays a 1-dimensional array that represents the value of the variable on each node. If you are printing a 1-dimensional array, Prism uses a 2-dimensional visualizer.

For C and C* programs, axis 0 represents the nodes. For Fortran 77 and CM Fortran programs, the highest-numbered axis represents the nodes.

Prism can aggregate data from multiple nodes only if the expression has the same size and number of dimensions on each node; if it doesn't, Prism prints an error message. This is is typically only an issue in global/local programs where the compiler has automatically broken up a global CM array into different-sized pieces on each node. In this situation, you can issue the **whatis**

Copyright © 1991–1994 Thinking Machines Corporation

command to find out which nodes contain objects of the same size; you can then restrict your PN set to those nodes before printing.

In the example shown in Figure 46, the variable **board** is an 8x8 array (representing a chess board); the current PN set contains four nodes. Therefore, Node Prism displays a 3-dimensional visualizer. Axis 0 represents the nodes. The figure shows the values of **board** in the first node in the set. You would drag the white bar in the slider portion of the data navigator to display the values in the other nodes in the set. (Note that, for a 2-dimensional CM Fortran or a Fortran 77 array, where axis 3 would represent the nodes, you might want to rearrange the display axes so that axis 3 is on the slider. You can do this by clicking in the box to the left of the slider and changing the number to a 3.)

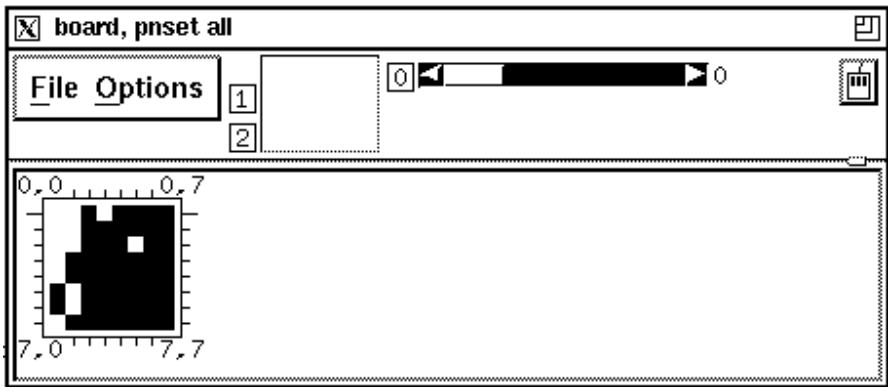

Figure 46. A visualizer in Node Prism (threshold representation).

To find out the value and node number for an element, shift-click on the element.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

Printing to the history region, or in commands-only Prism, works the same way. Axis 0 represents the nodes. Here is some of the history-region output for the data shown in Figure 46:

```
(prism all) print board
board =
PN 0
(0,0,0:4)  4          1          0          3          0
(0,0,5:7)  -1         0          -4
(0,1,0:4)  2          1          0          0          0
(0,1,5:7)  0          -1         0
(0,2,0:4)  3          1          0          0          0
(0,2,5:7)  2          -1         -3
(0,3,0:4)  5          0          0          0          -1
(0,3,5:7)  0          0          -5
(0,4,0:4)  4          0          0          -2         0
(0,4,5:7)  0          0          -6
(0,5,0:4)  0          1          0          0          0
(0,5,5:7)  0          -1         0
(0,6,0:4)  0          1          0          0          0
(0,6,5:7)  0          -1         0
(0,7,0:4)  6          -3         0          0          0
(0,7,5:7)  0          -1         -4
PN 1
(1,0,0:4)  4          1          0          3          0
(1,0,5:7)  -1         0          -4
(1,1,0:4)  2          0          0          1          0  ...
```

Note that the elements of axis 0 do not necessarily correspond to the numbers of the nodes they represent. For example, if you were visualizing a variable in PN set (1, 3, 5, 7), element 0 of axis 0 would represent node 1, element 1 would represent node 3, etc.

Node Prism provides a **Cycle** visualizer window you can use to display the values of a variable in the **`cycle`** PN set; see Section 10.3.8. If you issue the command

```
print x on cycle
```

Prism displays a window containing the value of **`x`** in the current node of the current PN set. If you then issue the **`cycle`** command or otherwise cycle

Copyright © 1991–1994 Thinking Machines Corporation

through the members of the **cycle** PN set, this window automatically updates to display the value of **x** in the next member of the set. This provides a convenient way of examining a variable in a series of nodes.

10.7 Performance Analysis in Node Prism

See Chapter 6 for general information on performance analysis in Prism.

Performance analysis in Node Prism is available for C* and CM Fortran programs running on a node. (You can use the C* or CM Fortran compiler to compile pure C or Fortran 77 programs, and thereby generate some of the data described below, but the timings will be different from those obtained with the native compilers.)

Node Prism performance analysis provides data only for activity taking place within a node. It does not measure activity taking place between the nodes and the partition manager, and does not trace or connect communication events between nodes.

As in Global Prism, you must compile your program with the **-cmprofile** option to collect performance data. To start collection, choose **Collection** from the **Performance** menu or issue the **collection on** command.

At the end of the run, you display the performance data by choosing **Display Data** from the **Performance** menu. The data is displayed for the current PN. If you change the current PN, as described in Section 10.3.7, the **Performance Data** window updates to show the data for the new PN.

Node Prism measures these resources, as shown in Figure 47:

- **Scalar cpu** — This is CPU time used by the scalar portion of the program running on the node. It is entirely CPU time used on the node's microprocessor. **NOTE:** Per-source-line data is not currently available for this resource.

  This resource includes time spent on the node by routines that weren't compiled with the **-cmprofile** option — for example, time spent in CMMD, CMSSL, and DPEAC routines. This data is available for C* programs as of Version 7.1.1 and for CM Fortran programs as of the final release of Version 2.1.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

- **Array cpu** — This is CPU time used for array operations in the program. It includes mostly vector unit CPU time, plus some startup CPU time on the node's microprocessor.

- **Array NEWS** — This is the time spent in NEWS operations (also referred to as *grid* communication) by the program within the node. (In the CM Fortran and C* on-a-node programming model, this is a memory move of one array into another.)

- **Array Send/Get** — This is the time spent in sends and gets (also referred to as general communication) by the program within the node. (In the CM Fortran and C* on-a-node programming model, this is a memory move of one array into another.)

- **Array Reductions** — This is the time spent doing data reductions within the node.

- **Scalar <–> Array** — This is the time spent in communication between scalar and array operations within the node.

- **I/O** — This is the time spent in I/O between the node and I/O devices. (This data is available as of CMMD Version 3.2.) **NOTE:** Per-source-line data is not currently available for this resource.

- **Total** — This is the total time spent in processing on this node. **NOTE:** Per-source-line data is not currently available for this resource.

Copyright © 1991–1994 Thinking Machines Corporation

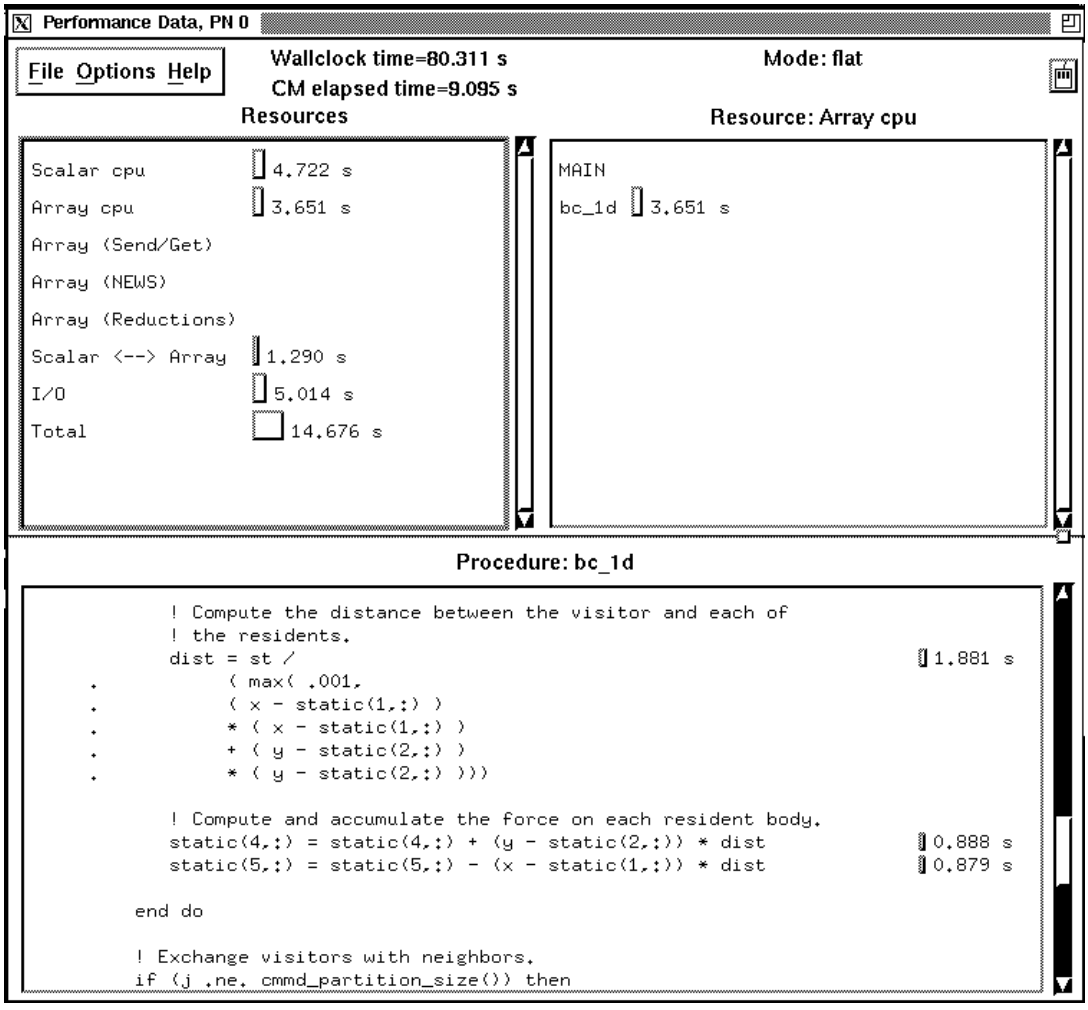

Figure 47. Performance data in Node Prism.

Users of Global Prism should note that these resource categories are different from those reported in that version of Prism; for example, Node Prism does not separate the resources into different subsystems. The new categories reflect the somewhat different execution model for a program running on a node.

Copyright © 1991–1994 Thinking Machines Corporation

10.7.1 Interpreting the Data

In general, you can interpret Node Prism performance data the same way you interpret the data in Global Prism; see Section 6.6. But note these points:

- There aren't separate subsystems, as there are in Global Prism, so you don't have to consider which subsystem is more heavily used.
- Section 6.6 talks about isolating bottlenecks by determining what source lines and procedures made the most use of the most heavily used resource. You can use the same process in interpreting Node Prism data, but be aware that the interpretation is less exact, because of the way message passing programs operate.

  There will be cases, for example, in which reducing the use of a resource in the instance of a program running on an individual node will not speed up the program as a whole, because the reduced usage occurs before a barrier synchronization, and the node is not on the critical path for reaching this synchronization point. Conversely, the node that *is* on the critical path may be using a resource at that time that is not the most heavily used resource over the entire run of the program. Reducing the use of this resource may significantly improve the program's performance.

10.7.2 Saving and Loading Performance Data

You can save and load performance data as you do in Global Prism; see Section 6.7. Node Prism saves the performance data only for the current PN of the current PN set. When you load this data, the **Performance Data** window displays the filename of the file in which you saved the data.

If you want to save the performance data for another node, change the current PN to that node; its data will then be displayed in the **Performance Data** window. You can then save the node's data in a separate file. See Section 10.3.7 to learn how to change the current PN.

Copyright © 1991–1994 Thinking Machines Corporation

10.8 Customizing Node Prism

You can customize Node Prism just as you customize Global Prism; see Chapter 9. Changes you make in one apply to the other. Both Node Prism and Global Prism use the same **.prisminit** file. This could lead to error messages if you bring up Global Prism and the file contained commands specific to Node Prism. Therefore, Prism lets you specify that commands in your **.prisminit** file are for Node Prism only by bracketing them with **#ifdef NODE** and **#endif**. For example,

```
alias c cont
#ifdef NODE
pnset 0
alias c "cont; wait every"
#endif
```

These commands alias **c** to different things in Global Prism and Node Prism, and set the initial PN set to 0 in Node Prism.

To provide this feature, Prism must preprocess the **.prisminit** file; by default it does not do this. To tell Prism to preprocess the file, use the Prism resource **Prism.cppPath**, specifying the path to your C preprocessor as its setting; typically this is **/lib**. Thus, you would set the resource as follows:

```
Prism.cppPath:/lib
```

See Section 9.4 for information on setting Prism resources. Note, however, that commands-only Prism is not aware of the settings of Prism resources such as **Prism.cppPath**, unless the settings are contained in the system-wide Prism **app-defaults** file.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation