The
Connection Machine
System

CMMD User's Guide

Version 1.1
January 1992

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, October 1991
Revised, January 1992

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

About This Manual

Objectives of This Manual

The *CMMD User's Guide* is written for programmers who are writing or porting message-passing programs to run on the Connection Machine model CM-5. It

- Introduces the components and environment of the CM-5 system, as they are used for message-passing programming.
- Provides a brief description of the host/node programming model, and describes how that model is currently implemented on the CM-5.
- Introduces the tools currently provided on the CM-5 to assist in the development of message-passing programs.
- Provides, from time to time, a few "do's and don't's" for the successful creation and execution of message-passing programs on the CM-5.

This user's guide is intended to be used in conjunction with the *CMMD Reference Manual*, which describes the functions provided by the CM-5's message-passing library, CMMD. Both manuals assume that the programmer has some experience in writing message-passing programs in the language of his or her choice.

This edition of the manual documents Version 1.1 of the CMMD library and Version 7.1 of the CMOST operating system. The software it describes is still under development and may be subject to changes. As further developments and enhancements occur, the manual will be updated to reflect them.

Notation Conventions

The table below displays the notation conventions observed in this manual.

Convention	Meaning
bold typewriter	CMMD functions and UNIX and CM System Software commands, command options, and filenames, when they appear embedded in text. Also programming language syntax statements, and language elements such as keywords, operators, and function names, when they appear embedded in text.
italics	Argument names and placeholders in function and command formats.
`typewriter`	Code examples and code fragments.
`% ` **bold typewriter** `regular typewriter`	In interactive examples, user input is shown in **bold typewriter** and system output is shown in `regular typewriter` font.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
uucp Electronic Mail:	ames!think!customer–support
Telephone:	(617) 234–4000 (617) 876–1111

Chapter 1

Introduction

This manual provides information on the CMOST operating system and its associated utilities for programmers who are interested in node-level, message-passing programming on the CM-5 supercomputer.

The current model for such programming on the CM-5 is the host/node model: one C or Fortran 77 program running on the host initiates and monitors a second program (in the same language) that runs on each of a number of nodes. The number of nodes is set by the size of the CM-5 partition (explained below); the library used for interprocessor communication — that is, for the message passing itself — is a CM library named CMMD.

1.1 Partitions

The CM-5 is a highly scalable parallel processing computer. The number of computational processors (or nodes) on a CM-5 ranges from fairly small to very large.

No matter what its size, however, a CM-5 provides for both space-sharing and timesharing.

- Space-sharing occurs when the system administrator partitions the CM-5, alloting so many nodes to one partition, so many to another. The system administrator also decides which users have access to a given partition.

  Administrators can change partition sizes or access rules as needed to meet the needs of their sites.

- All partitions run the CMOST operating system, an enhanced UNIX operating system. Therefore, timesharing is the natural state on all partitions.

Users of the CM-5 have access to all UNIX facilities that normally form part of the SunOS version of UNIX. In addition, they have access to special tools and utilities provided by CM software to facilitate parallel programming.

1.2 Software Versions Documented

This edition of the *CMMD User's Guide* documents tools and utilities that are part of Version 7.1 of the CM operating system, CMOST.

It does not document the `cc` or `f77` compilers, which you would use for compiling your message-passing programs; please see SunOS documentation for that information.

1.3 Using a CM-5 System

A CM-5 supercomputer is a massively parallel supercomputer. It contains tens, hundreds, or thousands of processors. These processors are divided into two categories: processing nodes and control processors.

Processing nodes (PNs) make up the vast majority of processors inside the CM-5 system. They are the processors that do the actual computations on parallel data, communicating with each other to share data as necessary. (System software occasionally refers to these processors as processing elements, or PEs.)

Control processors (CPs) manage the CM-5's processing nodes and I/O devices. These processors provide major OS services for the system, handling the system's user interface, its I/O and network interfaces, and its system administration and diagnostic interfaces.

A group of processing nodes under the control of a single control processor is called a partition. The control processor managing the partition is known as the *partition manager* (PM). In the host/node message-passing model, the partition manager is the host, while the processing nodes are — naturally — the nodes.

Interprocessor communication networks connect all processors, of both types, to provide rapid, high-bandwidth communication within and between processes.

1.3.1 The User's View

Figure 1 illustrates a sample CM-5 system as it appears to a user. This particular system has two partition managers, which have been named Mars and Venus. Each of these PMs is currently managing a partition of 256 nodes. The system also has control processors managing some I/O peripherals, and one that is dedicated as a system console, for the system administrator's use.

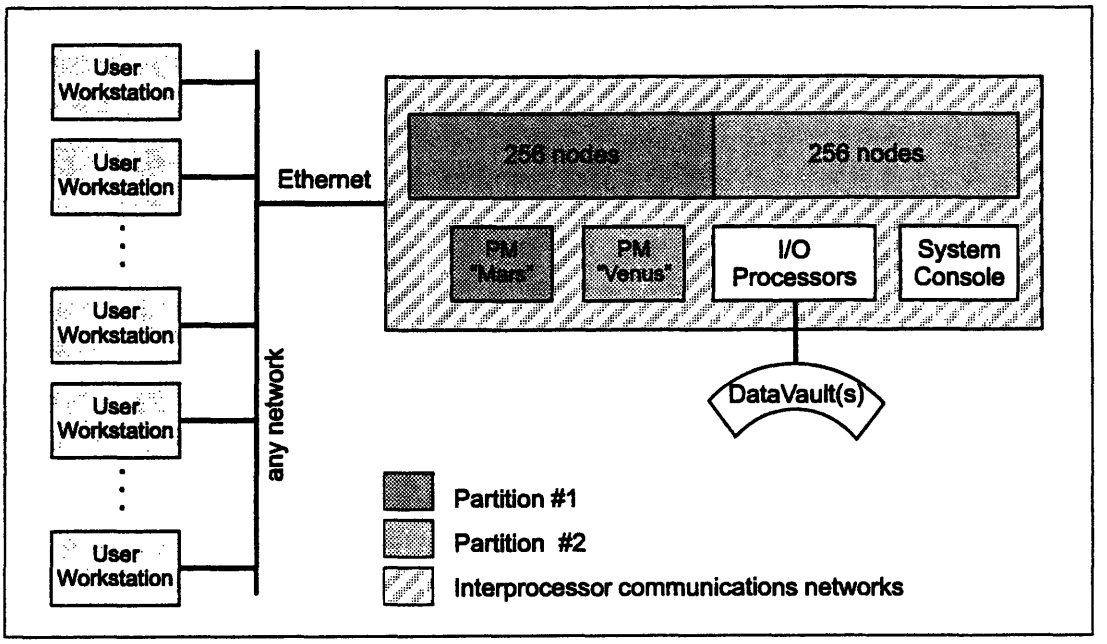

Figure 1. A sample CM-5 system.

Users (shown here at workstations "somewhere on the network") access the CM-5 system by accessing one of the PMs. They can log in remotely or use remote shells to run programs on either partition, assuming they have been granted access. Two examples might be:

```
% rlogin mars
<login sequence>
% a.out
```

```
% rsh venus my_program
```

A program runs on a single partition, using all the nodes on the partition for its parallel operations. If a program needs access outside the partition — to read

from an I/O device, for example, or to pass data to a process running on another partition — it goes through the partition manager to do so. (The partition manager, running in supervisor mode, can access any address in the system. The nodes, running the user's program in user mode, can access only addresses within their own partition.)

1.3.2 Keeping Up with System Status

Partitions are not permanent. They are defined by the system administrator to meet the site's needs, and can be changed as needed. The system shown in Figure 1, for example, could be reconfigured as a single partition, with Venus controlling all the parallel nodes and Mars either inactive or acting as a stand-alone compile server. Similarly, if some nodes needed to be taken out of service temporarily, the partition could be reconfigured around them.

The **cmps** command, given on a partition manager, tells users how many nodes the PM currently controls and what jobs it is running. For more information on this command (which is modeled after the UNIX **ps** command), see Chapter 4.

A Note to CM-2 Users:

Users familiar with the Connection Machine Model CM-2 will notice certain changes in the user environment on the CM-5.

- The commands that attached the front-end program to the parallel processors on the CM-2, **cmattach** and **cmcoldboot**, are not needed (and do not exist) on the CM-5 system. A CM-5 PM is always "attached" to its parallel nodes.
- The CM-2 informational commands **cmfinger** and **cmlist** are not available on the CM-5.
- The checkpointing facility is not yet available on the CM-5, but will be available later.

- The CM-5 hardware provides both SIMD and MIMD capabilities. Thus, users can write both data parallel programs and message-passing programs for the CM-5 system.*

  Data parallel programming uses the same CM parallel languages on the CM-5 as on the CM-2. Some libraries, such as the CM Fortran Utility Library and CMSSL, are identical (or nearly so) on the two machines. Others, such as graphics, are different.

  Message-passing programming uses standard C or Fortran 77 and is supported by the message-passing communications library, CMMD.

A Note to Users from Other Systems:

If you are porting message-passing programs from another system, you may notice several differences between their environment and the CM environment:

1. The CM-5 probably contains more nodes than any other machine for which you've written message-passing programs. Moreover, it will cheerfully let you write a message-passing program that uses every one of them.

   Programs ported from other machines, therefore, may spend different proportions of time computing and communicating. A program that ran on 8 or 16 nodes, and spent most of its time doing computation, may find itself much more evenly balanced between computation and communication when it runs on a considerably larger number of nodes. In some cases, this may make rethinking one's algorithms advisable.

* SIMD stands for Single Instruction, Multiple Data, and MIMD stands for Multiple Instructions, Multiple Data. These terms are sometimes used to describe programming styles as well as hardware. They are not, however, entirely accurate as software descriptors. Data parallel programming, as implemented on the CM-5 system, includes MIMD as well as SIMD capabilities, while message-passing programming often makes use of SIMD techniques.

2. You don't have to compete for nodes with other processes; the PMs run timesharing, so all partitions are timeshared. (The PM is the controlling processor here; when a host program that runs on the PM is swapped in or out, all its node programs are swapped in or out as well, host and node programs being treated as a single unit insofar as timesharing is concerned.)

3. On a CM-5, you don't need to allocate processors for your program: once you access a PM (via `rlogin`, `rsh`, or a batch command), you have automatically allocated all the nodes within the partition controlled by that PM.

   Therefore, your programs will always have nodes available to them, but they should plan on using all the available nodes.

4. The size of any given partition is configurable, and can change from day to day, or even from hour to hour. A PM that controls 128 nodes today may control 512 nodes tomorrow.

   You can, however, count on a partition size being a power of two: 32, 64, 128, 256, 512, etc. This may be of some help to you in planning your program. (For instance, it facilitates the use of trees within programs.)

   To take advantage of this flexible partitioning, your programs should make layout decisions at run time, so as to make optimal use of partitions of various sizes. You can use the `cmps` command (at shell level) and the `CMMD_partition_size` routine (within a message-passing program) to report the current partition size.

5. Because CMMD's global routines take advantage of hardware and software designed specifically for best performance when operating over all nodes in a partition, you will want to use these global routines whenever possible, rather than hand-coding your own such routines. (You will need to write your own routines if you want to do "global" operations, such as reduction, on a subset of the nodes in the partition.)

1.4 Why Use This Manual?

This manual describes the CMOST tools and utilities that support message-passing programming on the CM-5. It should be used in conjunction with the *CMMD Reference Manual*, which describes the message-passing library itself.

1.4.1 Software to Know About

This manual explains the following commands and tools, which are useful to programmers writing message-passing programs on the CM-5 system.

`cmps` The CM version of the UNIX **`ps`** command, **`cmps`** tells you how large a partition is and what processes are currently running on it. See Chapter 4, or the on-line man page.

`cmld` The CM version of the UNIX **`ld`** linker, documented in Chapter 3 of this manual and in an on-line man page.

`pndbx` The CM's node-level debugger, documented in Chapter 6 of this manual and the on-line man page.

`qsub` The NQS command by which users submit jobs for batch execution on the CM-5. See Chapter 4, or the on-line manual page, or *NQS for the CM-5,* Version 2.0.

`printf` The standard C "write" function to "standard output," invaluable in debugging. Since there are no terminals on the CM-5, **`printf`** sends its output to a file. See Chapter 5 for details.

CMMD timers
: Node-level timers, which allow timing of code running on individual nodes. See Chapter 4, or the on-line man page for any of the timer commands.

`CM_panic`
`CMPN_panic`
: CMOST calls that handle errors in host and node processes; by default, they halt the program, print an error message, and dump core. See Chapter 5, or the on-line man page.

core dump files
error files
: Files created by CMOST to help diagnose program errors, documented in Chapter 5 of this manual.

CMOS_SAFETY_LEVEL
: A CM environment variable that aids low-level debugging by enabling a record of the last few functions called by the dispatch loop.

rlogin
: The standard UNIX remote login command, used to log in to a CM partition.

rsh
: The standard UNIX remote shell execution command, used to run a program interactively on a CM partition without logging in.

1.4.2 Organization of the Manual

Within this user's guide,

- Chapter 2 describes the basic components of a message-passing program and explains what happens during program execution. It also provides a sample C program.
- Chapter 3 describes how to compile and link message-passing programs. It also describes a sample makefile that facilitates these tasks.
- Chapter 4 describes how to execute your programs. It also describes the CMMD timers and the **cmps** command.
- Chapter 5 describes some OS facilities for handling and diagnosing program errors.
- Chapter 6 describes how to debug message-passing programs using **pndbx** in conjunction with Prism or with **dbx**.

Chapter 2

Creating Message-Passing Programs

This chapter discusses the mechanics of creating a message-passing program, and describes briefly what happens when such a program runs on the CM-5. It also provides information on writing from the nodes, and concludes with a sample message-passing program.

The next chapter discusses compiling and linking message-passing programs.

2.1 Basic Components of a Message-Passing Program

Source code for a CM-5 program consists of three parts:

- Code to be run on the host processor (the PM).
- Code to run on the nodes.
- Interface code, which allows the PM to initiate node programs.

2.1.1 Host Code

Code that runs on the host (that is, on the partition manager) may contain anything ordinarily included in a program running on a Sun computer. This includes system calls, I/O calls, X11 routines, and calls to other specialized libraries.

In a message-passing program, code for the host must include calls that

- Enable the CMMD environment, to allow message passing.
- Invoke at least one node procedure, as a subroutine.
- Disable the CMMD environment when the program finishes.

The simplest host program, then, would look something like this:

```
#include <cm/cmmd.h>

void main()

{
    /* initialize CMMD */
    CMMD_enable();

    /* start node program running */
    node_runner();

    /* disable CMMD */
    CMMD_disable();

} /* main */
```

More often, however, host programs also include code that performs computations, makes CMMD calls to communicate with the nodes (perhaps to provide input or receive output from them), makes calls to other libraries or routines, or temporarily suspends message passing to allow calls to data parallel routines.

Section 2.2 provides further information on how host code performs these tasks.

2.1.2 Node Code

Code written for execution on a node consists of one or more possibly independent subroutines.

The subroutines may perform local computations and exchange data among node and host processors. At this release, they can do very little other I/O; see Section 2.3 for a discussion of node I/O. Similarly, many UNIX system calls are not supported on the nodes. If node programs invoke these unsupported calls, segmentation violations may ensue. The rule is, therefore: use node code for computation and for inter-node communication via CMMD library calls. Use the host (the PM) for external I/O and for system calls.

Node routines may be invoked

- From the host processor.
- From the nodes themselves.

The first routine must be invoked by the host. If it forms the only routine in the program (or at least the only one invoked from the host), it is equivalent to a "PN main" routine. Once the routine is running on the nodes, it can call other routines, as any program would. All the routines may be visible to the host.

It is also possible, however, for node code within a program to consist of a number of loosely related or even unrelated subroutines called from the host.

Routines that are to be invoked directly from the host must be named with a language-specific prefix.

- In C, the prefix is **`CMPE_`**
- In Fortran 77, the prefix is **`CMFPE_`**

2.1.3 Interface Code

Interface code provides special declaration code that allows the nodes to respond correctly to subroutine invocations from the host. It is generated with CM utility functions:

- For C, the function is **`c_sp_pe_stubs`**
- For F77, the function is **`f77_sp_pe_stubs`**

For information on how these files work, plus more information about how to write them, please see Section 2.2.2.

2.2 How a Message-Passing Program Runs

By now, the reader probably has a pretty good idea of the basic picture: one "main" program, which runs on the host, invokes as subroutines one or more programs that run on the nodes. These node programs do the bulk of the computation required by the overall program, passing their output back to the host as required.

But how does all this really work?

2.2.1 Initialization

The host initiates message-passing activity with a call to **CMMD_enable()**, which changes the state of the partition manager's network participation and initializes the message-passing environment.

NOTE

If any other CM activity has taken place, the user must ensure that it has finished before calling **CMMD_enable**.

After initialization, the host program is able to invoke node routines and call CMMD host-only and host-and-node routines.

2.2.2 Starting the Node Program

The steps in starting node programs are as follows:

1. When the user starts the overall (host) program on the PM, the CMOST operating system automatically loads the node code into each node's memory.
2. The nodes, with user code loaded, now wait in a "dispatch loop" for invocation information from the PM.
3. The host program calls **CMMD_enable** to initialize message passing.

4. The host program invokes the first node routine. Typically, this is a user-written routine with a name prefixed as described above, invoked via an ordinary subroutine call.

   By making this call, the PM broadcasts a message containing the address of the function to be invoked, the number of arguments to the function, and the argument values.

   Each node kernel receives the message, constructs an appropriate stack frame, and invokes the designated function. Since each node in the partition does this, the node code starts on all nodes.

Interface Files, Again

In order for Step 4, above, to happen, the PM must be able to transmit the right information and the nodes must be able to interpret the data correctly. This transfer of information is mediated through a special interface file.

An interface file consists of special declaration code that allows the PM to provide the nodes with the information to form the correct stack from the broadcast data. To make it easy for the user to generate the correct file, two utility functions, **`sp-pe-stubs`** and **`f77-sp-pe-stubs`**, have been provided.

For each node function that the host must see, you create a function prototype. In C, this is just an ANSI prototype. In Fortran the format is similar, but with Fortran types instead of C type names.

NOTE

1. Since the PM must broadcast the arguments of the interface functions to the nodes, these arguments cannot be pointers or structures. They must be standard C or Fortran types.

2. Even in Fortran prototypes, the code must be terminated with a semicolon.

You then place all the prototypes into one or more files (named, by convention, **`.proto`** files) and pass them as input to the utility function. Output files, by convention, use the suffix **`.intf.c`**.

Finally, you compile the output (`.intf.c`) files and include them in the link phase of the host code. See Chapter 3 for more information.

2.2.3 While the Program Runs

While the program is running, any given node may be in one of three states:

- Executing code.
- "Blocking," i.e., waiting within a code block for data or synchronization.
- "In the dispatch loop," waiting between procedures for further instructions.

Each node executes its code asynchronously, fetching data and instructions from its local memory. It synchronizes with other nodes only when required to do so for message-passing purposes (e.g., to send or receive a message, or to participate in a global instruction).

Each time the nodes finish a routine, they go back into the dispatch loop and wait for further instructions. Certain CMMD calls, such as barrier synchronization, global timing functions, and **`CMMD_suspend`**, can be given only while the nodes are in the dispatch loop. If such calls are intermixed with other message-passing calls — that is, if they are given during execution of a procedure — the program will hang.

2.2.4 Ending the Program

When the nodes have finished executing their part of the overall program, they return to the dispatch loop.

When all message-passing activity has been completed, the host program should invoke **`CMMD_disable`** to free internal CMMD memory allocation and return the host's network participation to its prior state.

2.2.5 A Few Caveats

1. Don't forget to use **CMMD_enable** and **CMMD_disable**. If you forget, CMOST will terminate your program with an unsightly error.

2. If your program hangs, in all likelihood someone is waiting for a message that has not been sent.

3. Allowing host code to become disordered so that the host calls for results (e.g., via **CMMD_reduce_from_nodes**) before invoking the node routine that contains the matching call (e.g., **CMMD_reduce_to_host**) is a surprisingly common method for achieving such program hangs.

4. If your program fails with the message:

   ```
   Ts-daemon failed to set up user memory on PE
   Error: Couldn't register with the TS daemon!
   ```

   it means your program requires more memory than is available on each node.

2.3 Writing from the Nodes

Neither Fortran I/O commands nor (with one exception) C I/O commands are currently available on the nodes. The exception is the **printf** command, discussed in Section 2.3.1.

CMMD calls, however, can be used to transfer data from the nodes to the host. The messages can come from any node or nodes, asynchronously, or from all nodes together (e.g., via **CMMD_concat_to_host** and **CMMD_gather_from_nodes**). Since the host program runs on the PM, it can use standard UNIX I/O calls to store or print data.

Using global calls to pass data to the host causes the node programs to synchronize. Since the more frequently a program synchronizes, the easier it is to debug, forcing synchronization through concatenation or reduction to the host of data or debugging messages (which can be removed as the program stabilizes) can constitute a handy debugging trick.

Using global calls in this manner may be faster than using **printf**. If a program has hit a communications deadlock, however, the global calls will also be dead-

locked. Calls to **printf**, on the other hand, go through the OS. They will therefore succeed even if the program itself is blocked.

2.3.1 Using printf

A node version of **printf** is available for debugging C programs. Its use will slow down your code. It writes output to a special file (as explained in Chapter 5) with the output from each node prefaced by **pn***XXX*, where *XXX* is the node identifier.

Fortran **print** and **write** will not work on nodes. If Fortran programmers wish to print from nodes, they must create their own specialized calls to **printf**. To do so requires a modest understanding of Sun interlanguage calling protocol:

1. Fortran passes arguments by reference. Therefore, a C routine called from Fortran must expect arguments that are pointers. Any return value must be passed as a value, not as a pointer.

2. C routines to be called from Fortran must have names that are entirely lowercase, and that end in a trailing underscore.

3. Fortran routines that call C programs should expect that floating-point return values will be double precision.

4. Fortran routines expecting return values from C code must be be declared to be of the appropriate type. Declaring them to be external is also useful.

An example:

To print the line "Hello world. x = 5", an appropriate wrapper would be:

C code:

```
#include <stdio.h>

void my_print_( string, value )
char *string;
int *value;
{
   printf("%s%d\n", string, *value );
}
```

The Fortran code then calls this routine as follows:

```
integer x
```

```
x = 5
call my_print( "Hello world.  x = ", x )
```

A somewhat more versatile approach would be to use the Fortran write statement to generate a string and then print the string via a wrapper. This would look like:

C code:

```
#include

void print_string_( string )
char *string;
{
  printf( "%s\n", string );
}
```

Fortran code:

```
      character*(80) output_string
      integer x

      x = 5

      write( output_string, 10) x
10    format( 1x, "Hello world. x = ", i1 )

      call print_string( output_string )
```

2.4 A Sample Program

The program **pi** uses a master/worker paradigm to perform a Monte Carlo approximation of pi. Points are randomly generated within a unit square centered at (0,0) and with vertices at (1,1), (–1,1), (–1,–1), and (1,–1). We count the number of "hits" as the number of points that are within distance of 1 to the origin. We can then approximate pi from the ratio of hits to the total number of points generated.

The program takes two arguments: *numtrials*, the total number of points to be generated, and *work_increment*, the number of points each worker should generate and test at a given time.

The PM is the master and the nodes are the workers. The workers send FREE_WORKER messages to the master to let it know that they are free. If there

is work to be done, the master then sends the first available worker a WORK message that tells that worker to generate and test a specified number of points. If a worker has finished its job, it will send the master an ANSWER message that reports the number of hits the worker counted. This entire process continues until there is no more work to be given to the workers.

The master then waits for any outstanding ANSWER messages before sending all workers a DONE message.

Below is a sample run in which 10 million points were generated with a *work_increment* of 1 million and 4 nodes available as workers.

```
% pi  10000000  1000000
Monte Carlo approximation of pi using 4 nodes to perform 10000000
trials
          Work increment is 1000000

          Worker 2 reports 785050 hits
          Worker 3 reports 785387 hits
          Worker 1 reports 785824 hits
          Worker 0 reports 784892 hits
          Worker 3 reports 785203 hits
          Worker 2 reports 784890 hits
          Worker 1 reports 785723 hits
          Worker 0 reports 785825 hits
          Worker 3 reports 785326 hits
          Worker 2 reports 785576 hits
          pi = 3.141478
task done
101.5u 0.5s 2:05 81% 0+464k 2+3io 12pf+0w
%
```

NOTE

Master/worker programs are normally used for tasks that require intensive computation on each node. The **pi** program, being designed simply to illustrate the basic shape and style of a master/worker program, performs a trivial task. Because the nodes have so little computational work to do, they spend much of their time idle. The moral? Copy this template only if your applica-

tion will keep the nodes well occupied with substantial computing tasks.

Here are the five files involved in **pi**:

1. The header file, **pi.h**

```
/*--------------------------------------------------------------------

 File: pi.h

Example program from CMMD User's Guide.

/*--------------------------------------------------------------------

/* message tags */

#define FREE_WORKER 0
#define WORK 1
#define ANSWER 2
#define STOP 3

/* auxiliary routines */

void print_usage(char *name);
void start_workers();
```

2. The prototype file: **pi.proto**

```
void start_workers();
```

3. The host code: **pi-pm.c**

```
/* --------------------------------------------------------------------

File: pi-pm.c

Contains the main program to be run on the partition manager to coordinate
"workers" on the processing nodes.  The program approximates pi by a Monte Carlo
method.

--------------------------------------------------------------------*/

#include <stdio.h>
#include <cm/cmmd.h>
#include "pi.h"
```

```
void
 main(int argc, char *argv[])
{
  int numtrials, work_left, work_increment, numhits=0, numworking=0, answer;

  if (argc<3)
    {
      print_usage(argv[0]);
      return;
    }

  numtrials = atoi(argv[1]);
  work_left = numtrials;
  work_increment = atoi(argv[2]);

  CMMD_enable();
  printf("Monte Carlo approximation of pi using %d nodes to perform %d
          trials\n", CMMD_partition_size(), numtrials);
  printf("\t Work increment is %d \n\n", work_increment);
  start_workers();

  while (work_left>0)
    {
      int free_worker;

      /* while there are workers free, send them work */
      if (CMMD_msg_pending(ANY_NODE, FREE_WORKER))
   {
     CMMD_receive(ANY_NODE, FREE_WORKER, NULL, 0);
     free_worker = CMMD_msg_sender();

     if (work_left < work_increment)
       {
         CMMD_send(free_worker, WORK, &work_left, sizeof(int));
         work_left = 0;
       }
     else
       {
         CMMD_send(free_worker, WORK, &work_increment, sizeof(int));
         work_left -= work_increment;
       }
     numworking++;
   }

      /* receive finished work from workers */
      if (CMMD_msg_pending(ANY_NODE, ANSWER))
   {
     CMMD_receive(ANY_NODE, ANSWER, &answer, sizeof(int));
     numhits += answer;
     printf("\t Worker %d reports %d hits \n",
        CMMD_msg_sender(), answer);
     numworking--;
   }
    }
```

```
  /* Check that all workers have finished */
  while (numworking>0)
    {
      if (CMMD_msg_pending(ANY_NODE, ANSWER))
   {
     CMMD_receive(ANY_NODE, ANSWER, &answer, sizeof(int));
     numhits += answer;
     printf("\t Worker %d reports %d hits \n",
        CMMD_msg_sender(), answer);
     numworking--;
   }
    }

  /* Stop all workers */
  {
    int worker;

    for (worker=0; worker < CMMD_partition_size(); worker++)
      {
   CMMD_receive(worker, FREE_WORKER, NULL, 0);
   CMMD_send(worker, STOP, NULL, 0);
      }
  }

  CMMD_disable();

  printf("\t pi = %f \n", ((float)(4*numhits)/(float)(numtrials)));
  printf("task done \n");
  exit(0);
}

void
 print_usage(char *name)
{
  fprintf(stderr, "Usage: %s <number of trials> <work increment>\n", name);
}
```

4. The node code: `pi.pe.c`

```
/* ------------------------------------------------------------------

 File: pi.pe.c

Contains the "worker" code which runs on the processing nodes to calculate pi
via the Monte Carlo method.  The principal routine is CMPE_start_workers which
is called from the main program running on the partition manager.

------------------------------------------------------------------*/

   File: pi-pe.c

#include <cm/cmmd.h>
#include <math.h>
#include "pi.h"
```

```
/* Monte Carlo trial for calculating pi.
   A point (x,y), -1 < x,y < 1 is randomely generated.
   Returns TRUE if the distance from (x,y) to (0,0) is less
   than or equal to 1. */

int approx_pi_points()
{
 float x=2.0, y=2.0;

 while ((x < -1) || (x > 1))
  x = 1 - ((float)(random() & (long)((1<<20)-1)))/((float)(1<<18));

 while ((y < -1) || (y > 1))
  y = 1 - ((float)(random() & (long)((1<<20)-1)))/((float)(1<<18));


 return sqrt(x*x + y*y) <= 1;
}

/* worker code */
void CMPE_start_workers()
{
 int work;        /* number of trials to perform */

 srandom(CMMD_self_address());
 CMMD_send(CMMD_host_node(), FREE_WORKER, NULL, 0);
 CMMD_receive(CMMD_host_node(), ANY_TAG, &work, sizeof(int));

 while (work)
  {
   int i, hits=0;

   switch (CMMD_msg_tag())
    {
    case WORK:
     for (i=0; i<work; i++)
      if (approx_pi_points()) hits++;
     CMMD_send(CMMD_host_node(), ANSWER, &hits, sizeof(int));
     CMMD_send(CMMD_host_node(), FREE_WORKER, NULL, 0);
     CMMD_receive(CMMD_host_node(), ANY_TAG, &work, sizeof(int));
     break;

    case STOP:
     return;

    default:
     printf("Error: bad tag %d \n", CMMD_msg_tag());
     return;
    }
  }
}
```

5. The makefile (see Chapter 3 for an explanation of this file):

```
# Makefile for pi -- a master/worker CMMD example
# that performs a Monte Carlo approximation of pi.

# for dependency checking
.KEEP_STATE:

MAIN            = CC

TARGET          = pi
SP_SRCS         = pi-pm.c
PE_SRCS         = pi-pe.c
C_PROTO         = pi.proto

INCDIR          =
USR_LIBDIR      =
CC              = gcc
OPT_LEVEL       = -g

# USR_SP_LIBS   =

# USR_PE_LIBS   =

#include file that does all the work
include /usr/include/cm/cmmd.make.include
```

##########

Chapter 3

Compiling Your Code

Compiling and linking of CMMD programs are best done using CMMD's makefile. They can, however, be done in separate steps. This chapter describes each step as it would be done separately, and then describes and explains the use of the makefile.

3.1 Compiling Your Code

The compilers used for message-passing programs are `cc`, `gcc`, and Sun `f77`.

Each type of code — host code, node code, and interface code — must be compiled separately from the other types. (For definitions of these types of code, see Chapter 2.)

CMMD header files are common to both host and node code. For C, the header file is `cmmd.h`. For Fortran, it is `cmmd_fort.h`.

3.2 Linking Your Code

Use the `cmld` linker to link object code for the CM-5. `cmld` is a version of the standard Sun linker that has been extended to accept host and node objects and to produce from them a single executable file. The linker accepts all the flags accepted by the Sun linker as well as the `-pe` marker flag, which separates host and node objects.

To link a C program, use the following syntax:

```
% cmld <host flags, paths> <host objects> <interface objects>
      <host libs> -pe <node flags, paths> <node objects> <node libs>
```

You can use the **-o** flag, either as a host flag or as a node flag, to specify the name of the executable file. (If you use the **-o** flag in both places, the name you specify with the node flag is used.)

To link Fortran code, use the CM Fortran compiler driver. This ensures that a Fortran main routine is linked in.

For example:

```
% cmf -cm5 myfile.o myfile.intf.o -lcmmd -lcmna_sp-lm
          -pe myfile.pe.o -lcmmd_pe -lcmna_pe -lm
```

3.3 Libraries

CM-5 libraries are provided for both the host code and the node code. Host libraries are identified as *libname*.**a** (except for the CMNA library, **cmna_sp.a**). Node libraries are identified as *libname*_**pe.a**. All libraries provided by Thinking Machines Corporation may be found in **/usr/lib**.

Programs using CMMD must link with the following libraries, all of which are provided by the makefile discussed in the next section:

For the host code:	For the node code:
cmmd	cmmd_pe
cmna_sp	cmna_pe
m	m

3.4 Compiling and Linking with a Makefile

To make compiling and linking easier, Thinking Machines provides a makefile to compile and link your message-passing programs. This makefile consists of two parts: the specific makefile and the **cmmd.make.include** file.

In the specific makefile, you assign program-specific values to the following make variables:

MAIN = Either F77 or CC. This indicates which main routine to link into the executable.

TARGET = The name of the executable.

SP_SRCS = The names of the host source files.

PE_SRCS = The names of the node source files.

C_PROTO = The names of the C prototype files.

F77_PROTO = The names of the F77 prototype files.

INCDIR = A list of directory paths to add to the header search path.

USR_LIBDIR = A list of directory paths to add to the linker search path.

CC = The name of the C compiler.

OPT_LEVEL = The optimization level with which compilation is performed.

USR_SP_LIBS = A list of additional libraries to provide to the linker for the PM link phase. NOTE: **cmmd**, **cmna_sp** and **m** are already included.

USR_PE_LIBS = A list of additional libraries to provide to the linker for the node link phase. NOTE: **cmmd_pe**, **cmna_pe** and **m** are already included.

Finally, you include the common **cmmd.make.include** file into the makefile.

```
include /usr/include/cm/cmmd.make.include
```

The **cmmd.make.include** file uses the values of the preceding variables to create the interface files, compile the host, node, and interface sources as indicated, and then link the host, interface, and node objects into the appropriate executable files.

The makefile then cleans up after itself. It removes intermediate files (like **.intf.c** files), and creates a **make.depend** file for complete dependency checking.

Several options allow you to request additional tasks:

clean indicates that **make** should remove all object files, depend files, and depend backup files.

echo indicates that the values of selected make variables should be displayed.

env indicates that the values of all environment variables should be displayed.

print indicates that the make variable specified by the input variable NAME should be displayed. This is useful for debugging make errors. For example:

```
make -f Makefile NAME=FOO_FILES print
```

would print the value of **FOO_FILES** as determined by the make program.

3.5 A Sample Makefile

As an example of a makefile, here is the makefile for **pi**, the sample program shown in Chapter 2. (The **cmmd.make.include** file, which is standard for all programs, is provided on-line.)

```
# Makefile for pi -- a master/worker CMMD example
# that performs a Monte Carlo approximation of pi.

# for dependency checking
.KEEP_STATE:

MAIN            = CC

TARGET          = pi
SP_SRCS         = pi-pm.c
PE_SRCS         = pi-pe.c
C_PROTO         = pi.proto

INCDIR          =
USR_LIBDIR      =
CC              = gcc
OPT_LEVEL       = -g
```

```
# USR_SP_LIBS    =

# USR_PE_LIBS    =

#include file that does all the work
include /usr/include/cm/cmmd.make.include
```

3.6 A Sample Make Session

Here is a sample session using the makefile shown above:

```
% make
gcc -g -DCM5 -DMAIN=main pi-pm.c -c  -o pi-pm.o
gcc -g -DCM5 -Dpe_obj -DPE_CODE  pi-pe.c -c  -o pi-pe.pe_o.o
mv  pi-pe.pe_o.o pi-pe.pe_o
/usr/bin/c_sp_pe_stubs < pi.proto > pi.intf.c
gcc -g -DCM5 -DMAIN=main  -c pi.intf.c
/usr/bin/cmld -o pi pi-pm.o  pi.intf.o  \
        -lcmmd -lcmna_sp -lm \
-pe pi-pe.pe_o \
        -lcmmd_pe -lcmna_pe -lm
```

Chapter 4

Executing Programs

This chapter discusses

- Checking system status.
- Executing programs interactively.
- Submitting batch jobs.
- Timing programs.
- Printing output.

4.1 The Execution Environment

The program execution environment on the CM-5 is similar to that of any UNIX system, with enhancements to handle parallel processing.

As with any system, you

- Gain access.
- Perhaps check system status.
- Run your program.

4.2 Gaining Access

To gain access to a CM-5, you must know the name of one or more of its partition managers. In addition, you must have been granted access rights by the system administrator.

The CM-5 is usually accessed across a network, either by logging in remotely (via the UNIX **rlogin** command), by running a remote shell (via the **rsh** command), or by submitting a batch job (via the **qsub** command).

Once you have logged in or established your shell, you are operating in the CMOST timesharing environment, with the following resources available to you:

- A partition manager (equivalent to a UNIX workstation). You initiate program execution on this processor, which utilizes parallel nodes and I/O devices as needed.
- All the parallel nodes in the partition. Under the CMOST timesharing environment, all the nodes are available to, and used by, all the parallel programs running on that partition.
- All the I/O devices on the CM-5 (assuming the system administrator has granted you access to the appropriate file systems).

4.3 Checking System Status

The two most common questions about system status on a CM-5 are

- How large is this partition at this time?
- How many users are running on it?

You can use the **cmps** command (modeled after the UNIX **ps** command) to answer these questions. The **cmps** command provides information about the partition on which the command runs. If you're logged on to Mars, the command **cmps** provides information on Mars. To find out about conditions on Venus, you would use a remote shell and type **rsh venus cmps**. In either case, the **cmps** output would look something like this:

```
% cmps

4 PN System, 0xb5e000 memory free, 3 Processes, Daemon up: 15:15

USER     PID  CMPID     TIME  TEXT+DATA  STACK    PSTACK   PHEAP  COMMAND
sal      4722   1      14:51 0025000    000c000  0310000 0000000 radix
kim      4949   2      11:51 004f000    000c000  0050000 0000000 get.hw
jan      *6111  3       3:14 0023000    000c000  0000000 0000000 nq
```

The first line of the **cmps** output provides general information about the partition, including the number of nodes (or PNs) it contains. The columns give information about each process.

The time column indicates the amount of time that the CMOST timesharing daemon has made available to the process, regardless of whether the process actually utilized the nodes. For timing information on how your program uses the nodes, use the timer functions described later in this chapter.

The memory columns, given in hexadecimal notation, refer only to the nodes. The *stack* is the UNIX process stack on each node, while *pstack* and *pheap* refer to memory allocated for user data. To find comparable data for the partition manager, use the UNIX **ps** command.

4.4 Executing a Program

The CMOST operating system treats the partition manager and its nodes as a single unit. Thus, you execute a message-passing program, or other parallel program, simply by executing the host program on the PM, as you would any UNIX program on any UNIX system:

```
% a.out
```

You can also execute a program in the background or by means of the **at** or **batch** command, as on any UNIX system, or via the NQS batch system's **qsub** command (described in the next section).

4.5 Executing a Batch Job with NQS

In a batch system, you submit one or more programs as a request to a queue. The batch system in turn submits the queued requests for execution. Your request is generally executed when it reaches the head of the queue. The CM system administrator is in charge of configuring queues to meet the needs of the site, and of informing users what queues are available when.

The CM batch system is based on the standard Network Queueing System (NQS).

NQS provides four user commands:

qsub	Submit a batch request.
qdel	Delete a batch request.
qstat	Display the status of queues and batch requests.
qlimit	Display the resource limits that can be placed on batch requests.

The following sections present a very brief introduction to the **qsub** and **qstat** commands. For full information on using the NQS batch system, please see *NQS for the CM-5*. You can also refer to the on-line manual pages for information on specific NQS commands.

4.5.1 Submitting a Batch Job

To submit a program for batch execution, you first create a *script-file*. A script-file is simply a file containing one or more program names. It may also contain instructions as to how NQS is to handle the program queueing and execution.

You then invoke NQS with the **qsub** command, and give it the name of the script-file. For example,

```
% qsub myscript
```

You can add options to the **qsub** command that supplement or override those in the script-file. For example,

```
% qsub -q mars1 myscript
```

This command line submits the script-file **myscript** to the queue **mars1**, no matter what queue the script-file specifies.

When your programs execute, output and error messages are written to files. By default, these files are placed in your current working directory. However, you can use **qsub** options to control their names and placement.

4.5.2 Checking on NQS

To find out the status of all your NQS requests, type

```
% qstat
```

To narrow your request to jobs on a specific queue, specify the queue name. To request status on all jobs (not just yours), use the **-a** option. Thus, to see the status of all jobs on queue **mars1**, type

```
% qstat -a mars1
```

For information on the queues themselves, use the **-b** option. See the on-line manual page and *NQS for the CM-5* for information on these options and on **qstat** in general.

4.6 Timing a Program

To time a message-passing program, insert calls to the CMMD timers within the program. These timers are much like the CM timers used to time data parallel code; but where those timers treat all the nodes as a unit, the CMMD timers treat each node separately. Each node calls its own timers, and each node's timers record times only for that node.

The paragraphs below summarize information about these timers.

4.6.1 Using the CMMD Timers

A program written with CMMD can use timers in either or both of two ways:

- A node can create and read timers for its own use, independent of any other node.
- The host processor can create timers on all nodes, and read maximum values from those timers.

Different sets of functions are provided for each model of use. Both sets follow the same pattern:

- First call **`timer_clear`** with an integer timer-ID to create a timer and initialize it to zero.
- Then call **`timer_start`**, to start the timer going.
- Then call **`timer_stop`**, information-reading functions, and **`timer_start`** whenever you like. Timings will be cumulative until **`timer_clear`** is called again.

A total of 64 timers is available to each node. Timers can be nested. Thus, one timer can provide timings for an entire routine, while a second timer provides timing for a particular block of code within the routine. Each timer can record timings of up to 43 hours, with microsecond precision.

Timers measure three values:

- *Busy time* is the time during which the user program is executing user code.
- *Idle time* is the time during which the user program is looping in the dispatch loop.
- *Elapsed time* is the sum of busy time and idle time. It represents the amount of time during which your process was scheduled for execution on the CM-5.

Timers give most accurate results when the program being timed has exclusive use of the partition. System load under timesharing can affect program timings.

4.6.2 Individual Timers, Called by Any Node

The following functions may be called by any node to manipulate timers for that node. No node can manipulate another node's timers, nor can the host manipulate timers initiated by a node.

Nodes can, however, cooperate in the use of timers. Moreover, a node can read a timer value into a buffer, then send that buffer to the host. If all nodes have read and stored timer values in an appropriate manner, they may perform a **reduce_to_host** to sum the time values, or to find the maximum or minimum value.

In all these functions, the value for *timer* must be an integer from 0 to 63, inclusive.

int CMMD_node_timer_clear(int *timer*)

Sets the total elapsed time, total CM busy time, and number of starts for *timer* to zero. Must be the first function called for any timer.

int CMMD_node_timer_start(int *timer*)

Starts the clock running for *timer*. Elapsed time and CM busy time are accumulated. Number of starts is incremented.

int CMMD_node_timer_stop(int *timer*)

Stops the clock running for *timer*. The specified timer's state variables for CM elapsed time and CM busy time are updated. A subsequent call to **CMMD_timer_start** — without an intervening call to **CMMD_timer_clear** — restarts the timer and adds to the accumulated elapsed and busy values for this timer.

double CMMD_node_timer_elapsed(int *timer*)

Returns the elapsed time recorded by this timer.

double CMMD_node_timer_busy(int *timer*)

Returns the busy time recorded by this timer.

double CMMD_node_timer_idle(int *timer*)

Returns the idle time recorded by this timer.

4.6.3 Global Timers, Called by the Host

The host uses the global timer functions to set and query timers running on all nodes simultaneously. There is no protection against a node manipulating one of these timers once the host has initiated it; such action, however, may well invalidate timings for the program as a whole.

Again, the value for *timer* must be an integer from 0 to 63, inclusive. It is the responsibility of the calling procedure to ensure that timers created by the host and those created by nodes have different identifying numbers.

Global timer calls operate through control blocks sent by the host to all nodes. Calls to these timers can therefore be made only while the nodes are in the dispatch loop between procedures. Hence, global timers can time entire procedures only; they cannot time code blocks within procedures. To time individual code blocks, use local timers on individual nodes, as explained above.

Global timer functions are as follows:

`void CMMD_clear_node_timers(int` *timer*`)`

Sets the total elapsed time, total CM busy time, and number of starts for *timer* to zero on all nodes. Must be the first function called for any global timer. Causes a global barrier synchronization.

`void CMMD_start_node_timers(int` *timer*`)`

Starts the clock running for *timer* on all nodes. Elapsed time (also known as wall-clock time) and CM busy time are accumulated. Number of starts is incremented. Causes a global barrier synchronization.

`void CMMD_stop_node_timers(int` *timer*`)`

On all nodes, stops the clock running for the specified timer and updates its state variables for CM elapsed time and CM busy time. Causes a global barrier synchronization.

A subsequent call to **`CMMD_timer_start`** — without an intervening call to **`CMMD_timer_clear`** — restarts the timer and adds to the accumulated elapsed and busy values for this timer.

`double CMMD_node_max_elapsed_time(int` *timer*`)`

Returns to the host the maximum elapsed time recorded on any of the nodes by the specified timer. Causes barrier synchronization.

double CMMD_node_max_idle_time(int *timer***)**

Returns to the host the maximum idle time recorded on any of the nodes by the specified timer. Causes barrier synchronization.

double CMMD_node_max_busy_time(int *timer***)**

Returns to the host the maximum busy time recorded on any of the nodes by the specified timer. Causes barrier synchronization.

Chapter 5

Error Handling and Error Diagnosis

There are several features built into the CM software that make debugging and error handling more convenient. The node-level debugger, **pndbx**, discussed in the next chapter, is one such feature. Various files that contain helpful information in the event of errors are another. Those files are discussed briefly in this chapter, along with the CM condition-handling routines, **CM_panic** and **CMPN_panic**.

5.1 Error Handling

The Connection Machine system provides two error handlers:

CM_panic ("*error_message*") for host programs

CMPN_panic ("*error_message*") for node programs

You can word your error messages to be as helpful as possible. Using different prefixes for different routines, for example, or otherwise identifying the source (and, to the extent possible, the cause) of the error is often useful, especially if you are writing code for others to use.

NOTE

For C programs, **CMPN_panic** is the cleanest way to terminate a program running on the nodes. Calling **EXIT** is valid only for programs running on the host, not for programs running on the nodes. (In the current implementation, calling **EXIT** from a node program produces a call to **CMPN_panic**. This behavior is not guaranteed to persist.)

For Fortran programs, a call to **STOP** by a node program produces an OS error and thus stops the overall program with an error message pointing to the code that called **STOP**.

5.1.1 Default Error Handling

The default behavior for both routines is to abort the currently running process, after printing the specified error message to the user's **stderr** and producing core dumps for the node and host processes. Both routines use the PM and the timesharing daemon to do this. (For details of the default behavior, see the **CM_panic(1)** man page.)

If you are running your code in the debuggers (Prism or **dbx** for the host code, plus **pndbx** for the node code), the debuggers will trap the error signal and halt your code. This allows you to examine and analyze the state of the failed program. (See Chapter 6 for information on using the debuggers.)

5.1.2 Customized Error Handling

You can alter the default behavior of **CM_panic** and **CMPN_panic** in several ways:

- You can set the environment variable **CM_NO_PN_CORE**, to disable the creation of the errors file and the node core dumps, stack file, and heap file. (The command line

```
setenv CM_NO_PN_CORE
```

accomplishes this.)

- The default behavior of both routines culminates in the host process receiving a **SIGTERM** signal. You may choose to install a different error handler for **SIGTERM**, or, alternatively, to have your program ignore the **SIGTERM** signal. (If the signal is ignored, the **CM_panic** routine simply returns; the program may or may not be able to recover.)

Please note that only wizards should try these tricks. They should consult the man pages for **CMOS_abort** and **CM_longjmp**.

5.2 When Your Program Is Terminated

If your program is terminated by a **SIGTERM** signal, you will usually get at least two things: PN (node) core dump(s), and a PN errors file. In addition to those files, Fortran programs may also get a Fortran traceback. The data contained in these three files, combined with some well-placed **printf** statements in your code, should help you to track down the cause of the error.

5.2.1 Using printf

If the nodes call the **printf** routine to print out data, the output is stored into the file **CMTSD_printf.pn.***pid* in the current directory (where *pid* is the process ID of your program). Using **printf** will slow down your program a great deal, so it is best used only for debugging.

Note that **printf** uses supervisor facilities, so that it will work even if your program has clogged the network.

5.2.2 The Errors File

In the directory from which you executed your program, you should find a file called **CMTSD_errors.***pid*. This 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 PM, if an error was detected there). The errors file will tell you which nodes crashed, and give you some information about the crash, such as what memory address the node was trying to reference, whether it died because of a segmentation fault, and so on.

5.2.3 Core Files

You should also find one or more node core files. These files are named **CMTSD_core_pn***X*.*pid*, where *pid* is again the process ID, and *X* is the node identifier. In some circumstances, you may also see a regular core file, from the host process.

If you don't want PN core files generated (generating them does take time), set the environment variable **CM_NO_PN_CORE** to any non-null value.

5.2.4 CMTSD Files

You may also see two files called **CMTSD_heap**.*pid* and **CMTSD_stack**.*pid*. These files contain the contents of the parallel stack and heap for the failed process. They are unlikely to be of much use to you. You can simply delete them, if you wish.

5.3 More about Cores

When an error occurs, cores for some of the nodes are dumped. To avoid wasting disk 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.

5.3.1 Looking at Core Files

Node core files have names of the form **CMTSD_core.pn***X*.*pid*, where *X* is the number of the node. You may examine the node core files with **dbx**. This requires two steps:

1. Use the **cmsplit** command to separate the host and node executables. The host executable will have the same name as the merged executable (e.g., **a.out**); the node executable will have a **.pn** suffix (e.g., **a.out.pn**). The syntax for **cmsplit** is

   cmsplit *program–name* [**-o** *node–file–name*.**pn**]

2. Invoke **dbx** on the node executable, supplying the name of the core file as an argument, as shown in the following example:

```
% dbx myprog.pn CMTSD_core.pn2.1234
warning: cannot read pcb in core file:
    registers' values may be wrong
Reading symbolic information...
Read 1885 symbols
program terminated by signal SEGV (segmentation violation)
   [or some other error]
(dbx) where
   [dbx tells you where it crashed...]
```

Note that you must give **dbx** the name of the node executable, rather than the host executable, to look at node cores. Specify the host executable to look at a host core file.

You can ignore the warning from **dbx** about registers' values. Node core files don't have some of the relatively obscure (undocumented) fields that normal UNIX cores have at the end, and **dbx** complains about this. It will not cause any problems with debugging.

5.4 Fortran Tracebacks: A Warning about Synchronization

When a Fortran program dies, it may generate a traceback. The traceback file will be called *prog*.**trace**, where *prog* is the name of your host program. The file is appended to every time your Fortran program dies, so if you crash multiple times, there will be multiple traces in the file. The last trace in the file is the newest one.

The traceback may give you an indication of which routine the code died in. However, the information may not be very valuable. Remember that the host and the nodes are not necessarily synchronized. The host frequently tells the nodes to begin some computation, then goes on to its own next task; therefore the host is frequently ahead of the nodes. Moreover, if a node has an error, the host may continue working for a while before the error status is propagated to it and your program halts. Therefore, the routine or instruction that is executing on the host when the nodes die may have nothing to do with the error.

Chapter 6

Debugging Your Program

6.1 Introduction

When you debug a message-passing program on the CM-5, you are actually debugging two programs — the host program and the node program — simultaneously but separately. There are two methods for doing this:

- You can debug your program inside the Prism programming environment. You use Prism's own windowed debugger to debug the host program and **pndbx** (invoked for you by Prism if you select **PN Debug** from the Utilities menu) to debug the node program. This is the preferred method, since Prism provides extensive debugging and data visualization features, as well as comprehensive on-line documentation.

- You can use the standard UNIX debugger, **dbx (1)**, to debug the host program, and the **pndbx** debugger to debug the node program.

Section 6.6 explains these two methods. Note that both use **pndbx** to debug the node program; **pndbx** is specifically designed for node programs.

The **pndbx** debugger has the same interface as **dbx**, with a few important extensions to handle parallelism. Because nodes may be operating asynchronously, **pndbx** works with one node at a time and allows the user to move among nodes at will.

For example, breakpoints are set on a per-node basis. You can set identical breakpoints on all nodes, or set different breakpoints for each node. However, you can see a particular node's breakpoints only if you have set that node as your current node.

The following discussion presents a brief overview of **pndbx**. Sections 6.2 through 6.4 list the features that **pndbx** provides for both high-level and

low-level debugging. Sections 6.5 through 6.9 discuss how to use **pndbx**. Section 6.10 provides an annotated sample debugging session.

The discussion in this chapter assumes that you are already familiar with **dbx**. If you have not used **dbx**, and you find the discussion here insufficient, please consult your SunOS or other UNIX documentation.

6.2 High-Level dbx Features Supported

This section lists **dbx** commands that are supported and extended in **pndbx**. Extensions are listed in Section 6.3.

6.2.1 The Essential Commands

The following list highlights key commands used in high-level language debugging. Note that these commands, when given in **pndbx**, apply only to the current node.

stop in *procedure*	Sets a breakpoint at the start of the specified procedure.
stop at *line*	Sets a breakpoint on the specified source line.
cont	Continues execution after being stopped by a breakpoint.
step, next	Single-steps into or over subroutines.
print *exp*	Prints the value of a variable or a source-language expression.
assign *var* = *exp*	Assigns a value to a variable.
where	Provides a stack trace.

6.2.2 Other Commands

Features of **dbx** that involve querying the symbol table or source file, such as **file**, **func**, **list**, **whatis**, and so on, are also supported. In general, these commands are not node-specific. See the on-line **pndbx** man page for details.

6.2.3 Commands Not Supported

Features of **dbx** that are inappropriate in the parallel context of a message-passing node program are not supported. These include tracing, watchpoints, and conditional breaks. Signal-handling control is also disabled.

6.3 Summary of Extensions

The **pndbx** utility extends the **dbx** command set by adding the following commands:

pn [*n*]	Identifies or changes the "current node," that is, the node to which node-specific **pndbx** commands refer. *n* is the node identifier for the node you wish to make current.
pnstatus [**all**]	Prints out the status of the current node, or of all nodes. Possible states are running, break, and error.
interrupt	Stops the current node and identifies the place in the source code at which the code was interrupted. NOTE: Many other **pndbx** commands, such as **where**, **print**, **stop in**, and **stop at**, also cause the current node to stop execution.
wait	Causes **pndbx** itself to wait (without displaying the **pndbx** prompt) until the current node reaches a breakpoint or encounters an error, thus notifying the user of the change in node status.

Three new arguments to **dbx** and **pndbx** commands also exist:

command **all**	Causes a **dbx** or **pndbx** command, such as **where** or **interrupt**, to operate on all nodes, rather than on just the current node.
command **stopped**	Causes a command to affect all stopped processes (that is, all those having a **pnstatus** of either break or error).
command **running**	Causes a command to affect all running processes.

These arguments apply to any commands for which they make sense. For example, you could request "**pnstatus all**"; "**where all**" or "**where running**" or "**where stopped**"; "**interrupt running**", or "**cont stopped**"; but you could not reasonably request "**pn all**".

6.4 Commands for Low-Level Debugging

Low-level debugging support in **pndbx** includes the following commands:

address [,*address*] **/ format**	Shows contents of a memory location (or range of locations).
address **/** [*count*] **format**	Shows contents of *count* memory locations, in a given format. Default count is 1.
print *register*	Shows contents of a register in hex.
register [,*register*] **/ format**	Shows contents of a register (or a range of registers), in a given format.
register **/** [*count*] **format**	Shows contents of *count* registers, in a given format. Default count is 1.
stopi at *address*	Sets a breakpoint at a code address.
stepi **nexti**	Single-step by machine instruction, either into or over calls.

assign *address* = *value*	Writes a value into a memory location.
assign *register* = *value*	Writes a value into a register.
number = **format**	Performs a radix conversion.

The formats for these commands are as follows:

d	**2-byte decimal**	**D**	**4-byte decimal**
o	**2-byte octal**	**O**	**4-byte octal**
x	**2-byte hex**	**X**	**4-byte hex**
f	**float**	**F**	**double**
i	**instruction**		

The default format is initially **X**. Specifying a format for any **pndbx** command, however, changes the default to the newly specified format. Thus, if you type "**1000/D**", you automatically set the default format to "**D**".

Register names are as follows:

$g0 - $g7	
$o0 - $o7	
$l0 - $l7	
$i0 - $i7	
$f0 - $f31	
$wim	window invalid mask
$psr	processor status register
$pc	program counter
$npc	next program counter
$y	Y register (step-multiply)
$tbr	trap base register
$sig	trap number that got the program into **pndbx**
$fsr	floating status register
$fq0	FP queue (address of pending FP instruction)
$fq1	pending FP instruction

NOTE

Printing **$fsr**, **$fq0**, and **$fq1** should be done with caution, as it may trigger a pending floating-point exception.

Some examples of legal commands are

`0x1000/10X`	Show ten hex words starting at virtual address hex 1000.
`print $pc`	Show current PC in hex.
`$pc/D`	Show current PC in decimal.
`0x2000/10i`	Show ten instructions starting at virtual address hex 2000.
`$g0/32X`	Show 32 registers, `$g0` through `$17`.
`1000=X`	Convert decimal 1000 to hex.
`0x3e8=D`	Convert hex 3e8 to decimal.

6.5 Compiling and Linking

If you intend to use **dbx** and **pndbx**, you must compile the source code for your host and node programs with **-g**. Then, when linking with **cmld**, pass **cmld** the **-lg** option, after the **-sp** or **-pe** flag.

6.6 Starting Up pndbx

How you start **pndbx** depends on whether or not you are using Prism. Using Prism makes the startup much simpler.

6.6.1 Using Prism

You can do much more with the Connection Machine's Prism programming environment than just debug programs. A simple debugging session, however, is all we will discuss here. For information on Prism itself and how you use it, see the *Prism User's Guide.*

Start by invoking Prism and giving it the name of the program you wish to debug. For example,

```
% prism hilbert
```

A Prism window appears on your screen, displaying the specified host program. You can set breakpoints and so on as you wish in this program.

In the Utilities menu, click on **PN Debug**. Then give Prism the **Run** command. When your program begins running, a new window opens. The new window contains **pndbx**, already invoked and active on your node program.

Alternatively, you can start executing your host program, let it run until it reaches a breakpoint, and then click on **PN Debug** to invoke **pndbx** on the node program.

6.6.2 Using dbx

To debug a program using **dbx** and **pndbx** you must first start up your host program under **dbx**, then invoke **pndbx** on the downloaded (and possibly running) node program. Usually, you run your host program under **dbx** in one window and your node program under **pndbx** in a second window.

How do you invoke **pndbx** if your program takes only a few seconds to run? You start up the host program under **dbx** and set a breakpoint in the code at a point just after the nodes have started executing. (The command "**stop in main**" accomplishes this.) When you hit the breakpoint, you may invoke **pndbx** on the node program, set any breakpoints, and then let the host program continue executing.

The command line for invoking **pndbx** is

pndbx *program–name host–pid*

For example:

```
% pndbx hilbert 14640
```

You can use the **cmps** command to get the process ID (**pid**) of the stopped host process. See the sample session at the end of this chapter for an example of this procedure.

6.7 Monitoring the Nodes

With **pndbx**, you monitor one node at a time. When you first start up, you are monitoring node 0. You can find out which node you are monitoring by using **pndbx**'s **pn** command. You can switch to a different node by using the **pn** *n* command, where *n* is the number of the node you want.

6.7.1 Asynchronous Monitoring

Because the nodes execute their programs asynchronously but simultaneously, **pndbx** is asynchronous with respect to the overall program being debugged. You can be typing commands at the **pndbx** prompt while some (or all) of the node processes being debugged are running.

Error handling in **pndbx** reflects this asynchronous operation. If one node encounters an error, that node goes into an error state and suspends execution at the point of the error. The other nodes, however, continue to execute the user program.

You can use **pndbx** to see which nodes are in an error or break state, switch to one of those nodes, and use debugger commands to see what is going on. If the node was in a break state (that is, if it was stopped because it hit a debugger-set breakpoint or was interrupted by the debugger), you can use the **cont** command to resume execution on the node.

6.8 Exiting from pndbx

The **quit** command in **pndbx** causes **pndbx** to exit. It also causes **pndbx** to clean up after itself by deleting all breakpoints on all nodes and continuing all stopped nodes.

A **quitfast** command also exists. This command causes **pndbx** to exit without cleaning up after itself.

6.9 Using pndbx

Like **dbx**, **pndbx** displays a prompt when it starts up. Unlike **dbx**, **pndbx** displays a prompt even when the current node is executing code. This is similar to running a process in the background from the shell. In general, you will always have a **pndbx** prompt, no matter what the node is doing. There are a few exceptions:

- The **step** and **next** commands do not display the prompt until the commands complete. They usually complete quickly, but sometimes they take a long time. When that happens, the prompt vanishes for a long time.
- The **wait** command (described later in this chapter) does not display the prompt until the next breakpoint is reached or an error occurs.

In any of these cases, typing Ctrl–C redisplays the prompt.

6.9.1 pnstatus

Since **pndbx** always displays a prompt, you need a way to find out whether the node is running, stopped at a breakpoint, or stopped with an error. You can find out what a node is doing by using the **pnstatus** command, which tells you the status of the current node. You can also use **pnstatus all** to find out the status of all nodes. (Note that this may take a minute or two on a large partition.)

NOTE

Because output for **pnstatus all** (and some other commands) may be long enough to scroll out of your window, or off your screen, **pndbx** paginates the output, printing a **more?** prompt after each 24 lines. You can change the pagination with the **pndbx** command

```
set $page_size = number–of–lines
```

Setting page size to 0 disables pagination and allows the output to scroll freely.

6.9.2 Interrupting Nodes

One important thing to remember when using **pndbx** is that many of the normal debugging commands you use (in particular, any command that reads or writes memory in the node) interrupt the nodes. When this happens, the nodes are not automatically restarted.

For example, if you want to find out where the current node is, and type **where**, you will get the expected information. After the command executes, the node remains stopped, regardless of whether it was stopped before you executed the command. You must explicitly type **cont** to let the node continue executing.

In general, therefore, you should be careful to check the status of a node after doing any **pndbx** commands, to be sure the node is in the state you think it is. If you forget to resume execution of a node, you (and the node) will simply sit there and wait, and nothing will happen.

You can also use the **pndbx** command **interrupt** to interrupt a node. This is similar to hitting Ctrl–C under regular **dbx**, to interrupt the process being debugged.

6.9.3 Waiting for Breakpoints and Errors

Because of the asynchronous nature of the debugger, no message is printed out when a node reaches a breakpoint. This could make it inconvenient to work with breakpoints, because you would not know if a node had reached its breakpoint unless you repeatedly used the **pnstatus** command.

To solve this problem, use the **wait** command. This command takes away the **pndbx** prompt. It causes **pndbx** to sit and wait until the current node reaches a breakpoint or encounters an error, at which time it restores the prompt.

To break out of a **wait**, hit Ctrl–C. This restores the **pndbx** prompt. Doing this will have no effect on the node; if the node is running, it will keep running.

There is currently no way to sit and wait until any node hits a breakpoint or error; you can only wait for the current node to do so.

6.10 A Sample pndbx Session

Assume two windows, one for debugging the host process with **dbx** and one for debugging the nodes with **pndbx**.

Start up the host process on the PM and execute to the start of "main":

Host Window

```
title% dbx hilbert
Reading symbolic information...
Read 2549 symbols
(dbx) stop in main
(dbx) run 1024 200 0 10 3
Running: hilbert 1024 200 0 10 3
stopped in main at line 116 in file "/users/title/cmmd/hilb.sc.c"
  116     second();
```

Start up **pndbx**:

Node Window

```
title% cmps
4 PN System, 0x6cd000 memory free, 2 Processes, Daemon up: 1 day, 14:04
USER       PID CMPID     TIME TEXT+DATA  STACK  PSTACK   PHEAP  COMMAND
amk      *14615   1      2:27  001b000  000c000 0000000 0000000 wa_dev1
title     14640   0      0:00  0047000  000c000 0000000 0000000 hilbert

title% pndbx hilbert 14640
Current partition size is 4
Welcome to pndbx version 1.0-beta of 8/2/91 9:54 (hespera.think.com).
Type 'help' for help.
reading symbolic information ...
(pndbx) pnstatus all
PN  pn-status
 0  running
 1  running
 2  running
 3  running
```

List some source code:

Node Window

```
(pndbx) func CMPN_pmain
(pndbx) list 408,418
  408     CMMD_receive_broadcast_from_host(&sqrts, sizeof(int));
  409     if (pe==0) printf("&sqrts = %x\n", sqrts);
  410     fillsqrts();
  411     if (pe==0) printf("fillsqrts\n");
  412
  413     /* Download primes and verify */
  414     CMMD_receive_broadcast_from_host(&primes, sizeof(int));
  415     if (pe==0) printf("&primes = %x\n", primes);
  416     CMMD_receive_broadcast_from_host(primes,
nprimes*sizeof(int));
  417     if (pe==0) printf("primes %d %d %d %d %d ...\n",
primes[0], primes[1],
 primes[2], primes[3], primes[4]);
  418     if (CMMD_self_address() == 0) {
```

Here we will set a breakpoint at line 408 in PN 0, a breakpoint at line 414 in PN 1, and a breakpoint at line 416 in all nodes. Note how as a side effect of setting a breakpoint, the nodes go into a "break" state. So after setting all the breakpoints, we continue all the nodes to get them back into the "running" state.

Node Window

```
(pndbx) stop at 408
interrupt in CMNA_dispatch_idle at 0xcb18
CMNA_dispatch_idle+0x78:        andcc   %i3, 16, %g0
[1] stop at "hilb.pe.c":408
(pndbx) pnstatus all
PN  pn-status
 0  break
 1  running
 2  running
 3  running
(pndbx) pn 1
(pndbx) stop at 414
interrupt in CMNA_dispatch_idle at 0xcb18
CMNA_dispatch_idle+0x78:        andcc   %i3, 16, %g0
[1] stop at "hilb.pe.c":414
(pndbx) pnstatus all
PN  pn-status
 0  break
 1  break
 2  running
 3  running
(pndbx) stop at 416 all
pn number 0:
[2] stop at "hilb.pe.c":416
pn number 1:
[2] stop at "hilb.pe.c":416
pn number 2:
interrupt in CMNA_dispatch_idle at 0xcb18
CMNA_dispatch_idle+0x78:        andcc   %i3, 16, %g0
[1] stop at "hilb.pe.c":416
pn number 3:
interrupt in CMNA_dispatch_idle at 0xcb18
CMNA_dispatch_idle+0x78:        andcc   %i3, 16, %g0
[1] stop at "hilb.pe.c":416
(pndbx) pnstatus all
PN  pn-status
 0  break
 1  break
 2  break
 3  break
```

Node Window (cont'd)

```
(pndbx) cont all
pn number 0:
pn number 1:
pn number 2:
pn number 3:
(pndbx) pnstatus all
(pndbx) pnstatus all
PN  pn-status
 0  running
 1  running
 2  running
 3  running
```

Note that the nodes remain "running" in the dispatch loop, and don't reach the breakpoints, because the host process is stopped at the start of "main." If we continue the host process past the CMMD startup code, the nodes can proceed:

Host Window

```
(dbx) list 159,170
  159     CMMD_sys_enable();
  160     pmain(N, NPRIMES, NLEG, DEPTH, XYLIM);
  161     printf("Primes to %d, Legendre entries %d.  ",
            primes[NPRIMES-1], legsize);
  162     fflush(stdout);
  163
  164     /* Allocate prodsqrt table */
  165     temp = CM_sbrk((N+1)*sizeof(int));
          /* Allocate heap for prodsqrt table */
  166     if ( temp == -1 ) {
  167       fprintf(stderr,"Not enough room on heap for prodsqrt
            table\n");
  168       exit(1);
  169     }
  170     CMMD_broadcast_from_host(&temp, sizeof(int));
          /* Tell PNs where it is */
(dbx) stop at 170
(4) stop at "/users/title/cmmd/hilb.sc.c":170
(dbx) c
Primes to 1223, Legendre entries 0.
  stopped in main at line 170 in file
  "/users/cmsg4/title/cmmd/hilb.sc.c"
  170     CMMD_broadcast_from_host(&temp, sizeof(int));
          /* Tell PNs where it is */
```

Now, going back to the node window, we see that node 0 has hit its breakpoint at line 408:

Node Window

```
(pndbx) pnstatus all
PN  pn-status
 0  break
 1  running
 2  running
 3  running
(pndbx) pn 0
(pndbx) where
[1] stopped in CMPN_pmain at line 408 in file "hilb.pe.c"
  408     CMMD_receive_broadcast_from_host(&sqrts, sizeof(int));
CMPN_pmain(nsearch = 1024, nprimes = 200, nleg = 0,
  depthlim = 10, xylim = 3), line 408 in "hilb.pe.c"
pe_main_default() at 0xca90
```

If we do a **next** command to step over the **CMMD_receive_broadcast_from_host** call, we hang because the host has not yet made the broadcast:

Node Window

```
(pndbx) next
```

So, switching back to the host, we'll step past where it does the broadcast. Along the way, we'll take a look at the value being broadcast:

Host Window

```
(dbx) print temp
'hilb.sc'main'temp = 0x80000000
(dbx) next
stopped in main at line 173 in file
    "/users/title/cmmd/hilb.sc.c"
  173     temp = CM_sbrk(NPRIMES*sizeof(int));
    /* Allocate heap for primes */
```

In the node window, the **next** now completes, and we check that the value received is correct:

Node Window

```
stopped in CMPN_pmain at line 409 in file "hilb.pe.c"
  409     if (pe==0) printf("&sqrts = %x\n", sqrts);
(pndbx) print sqrts
0x80000000
```

Checking node status again, we note that node 1 has hit its breakpoint at line 414.

Node Window

```
(pndbx) pnstatus all
PN  pn-status
 0  break
 1  break
 2  running
 3  running
(pndbx) pn 1
(pndbx) where
[1] stopped in CMPN_pmain at line 414 in file "hilb.pe.c"
  414     CMMD_receive_broadcast_from_host(&primes, sizeof(int));
CMPN_pmain(nsearch = 1024, nprimes = 200, nleg = 0,
  depthlim = 10, xylim = 3), line 414 in "hilb.pe.c"
pe_main_default() at 0xca90
```

Once again a **next** over the **CMMD_receive_broadcast_from_host()** hangs until we let the host continue execution:

Node Window

```
(pndbx) next
```

Host Window

```
(dbx) list
  174     if ( temp == -1 ) {
  175       fprintf(stderr,"Not enough room on heap for primes\n");
  176       exit(1);
  177     }
  178     CMMD_broadcast_from_host(&temp, sizeof(int));
            /* Notify PNs where it is */
  179     CMMD_broadcast_from_host(primes, NPRIMES*sizeof(int));
  180     CMMD_receive(0, DEFAULT_MSG_TAG, &temp, sizeof(temp));
  181     if (temp != primes[NPRIMES-1]) {
  182       fprintf(stderr,"Wrong max prime downloaded = %d\n", temp);
  183       exit(1);
(dbx) print primes
`hilb.sc`main`primes = 0x80001004
(dbx) stop at 180
(5) stop at "/users/title/cmmd/hilb.sc.c":180
(dbx) cont
stopped in main at line 180 in file "/users/title/cmmd/hilb.sc.c"
  180     CMMD_receive(0, DEFAULT_MSG_TAG, &temp, sizeof(temp));
```

Now we're unstuck in the node window, and can once again check that the node received the correct value for "primes":

Node Window

```
stopped in CMPN_pmain at line 415 in file "hilb.pe.c"
  415     if (pe==0) printf("&primes = %x\n", primes);
(pndbx) print primes
0x80001004
```

By now the remaining nodes have hit their breakpoints at line 416:

Node Window

```
(pndbx) pnstatus all
PN  pn-status
 0  break
 1  break
 2  break
 3  break
```

The **where all** command lets us see where all the nodes are stopped:

Node Window

```
(pndbx) where all
pn number 0:
CMPN_pmain(nsearch = 1024, nprimes = 200, nleg = 0, depthlim = 10,
  xylim = 3), line 409 in "hilb.pe.c"
pe_main_default() at 0xca90
pn number 1:
CMPN_pmain(nsearch = 1024, nprimes = 200, nleg = 0, depthlim = 10,
  xylim = 3), line 415 in "hilb.pe.c"
pe_main_default() at 0xca90
pn number 2:
[1] stopped in CMPN_pmain at line 416 in file "hilb.pe.c"
  416     CMMD_receive_broadcast_from_host(primes, nprimes*sizeof(int));
CMPN_pmain(nsearch = 1024, nprimes = 200, nleg = 0, depthlim = 10,
  xylim = 3), line 416 in "hilb.pe.c"
pe_main_default() at 0xca90
pn number 3:
[1] stopped in CMPN_pmain at line 416 in file "hilb.pe.c"
  416     CMMD_receive_broadcast_from_host(primes, nprimes*sizeof(int));
CMPN_pmain(nsearch = 1024, nprimes = 200, nleg = 0, depthlim = 10,
  xylim = 3), line 416 in "hilb.pe.c"
pe_main_default() at 0xca90
```

At this point, we'll allow execution to continue freely. We'll do a **continue** on the host side, and a **continue-all** on the node side. After the first **continue-all** on the node side, nodes 0 and 1 hit the line 416 breakpoint (since they haven't previously hit that breakpoint). A subsequent **continue-stopped** gets them past it:

Host Window

```
(dbx) c
```

Node Window

```
(pndbx) cont all
pn number 0:
pn number 1:
pn number 2:
pn number 3:
(pndbx) pnstatus all
PN  pn-status
 0  break
 1  break
 2  running
 3  running
(pndbx) cont stopped
pn number 0:
pn number 1:
```

Now all the nodes are running:

Node Window

```
(pndbx) pnstatus all
PN  pn-status
 0  running
 1  running
 2  running
 3  running
```

So our program runs to completion, as shown in the host window:

Host Window

```
Search on 4 PNs.
User-clock time from beginning of run=       16.34935 seconds.
c/n^2 = 219427/1048576 = 0.209261894226
User-clock time from beginning of run=       18.36593 seconds.
Master visit calls = 1243, subtasks = 1240
execution completed, exit code is 0
program exited with 0
```

And we exit both debuggers:

Host Window

```
(dbx) quit
```

Node Window

```
(pndbx) quit
```

Index

Symbols

A

B

C

D

Q

R

S

T

U

W

X