The
Connection Machine
System

CMMD User's Guide

Version 3.0

May 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, May 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1264
(617) 234-1000

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. The *User's Guide* does the following:

- It introduces the components and environment of the CM-5 system, as they are used for message-passing programming.
- It provides a brief description of the host/node programming model, and describes how that model is currently implemented on the CM-5.
- It introduces the tools currently provided on the CM-5 to assist in the development of message-passing programs.
- It provides a useful "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 3.0 of the CMMD library and Version 7.2 of the CMOST operating system.

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:

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

Chapter 1

Introduction

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

Two current models exist for such programming on the CM-5:

- The *host/node* model: a *host program* executing on the host starts up and monitors a *node program* running in multiple copies on each of a number of processing nodes. This programming model is currently supported for the C, C++, and Fortran 77 programming languages.

  Implementation Note: Currently, the host/node programming model is not supported in C*.

- The *hostless* model: a single node program executes in multiple copies on the processing nodes, while the host runs a CMMD-supplied I/O server program. This programming model is currently supported for the C, C++ and Fortran 77 programming languages, and for the parallel CM programming languages CM Fortran and C*, when executing in "on-a-node" style.

In both of these programming models, the number of processing nodes is determined by the size of the CM-5 partition in use (see below).

CMMD is the software library used for interprocessor communication — that is, for the message passing between nodes in the hostless model, and between host and nodes in the host/node model.

Message-passing programming is supported for the following languages: C, C++, and Fortran 77. Message-passing programming is also supported for the following parallel programming languages: CM Fortran and C*.

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, allotting 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 mode 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.

1.1.1 CM-5 Vector Units

In a CM-5 with vector unit accelerators (VUs) installed, each processing node has four associated VU accelerators. In this case, you have two options:

- You can run your code without the VUs (in any of the supported languages).

- You can run your code with VUs (in the parallel programming languages C* and CM Fortran, or by calling a DPEAC routine from C).

The VU accelerators provide enhanced performance for parallel programs that are largely arithmetic in nature.

1.2 Software Versions Documented

This edition of the *CMMD User's Guide* documents tools and utilities that are part of Version 7.2 of the CM operating system, CMOST.

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

1.3 Using a CM-5 System

The CM-5 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. (Note: On CM-5's with vector units, the four VUs associated with each node are considered part of the processing node.)

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 message-passing programming, 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 between processors.

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.

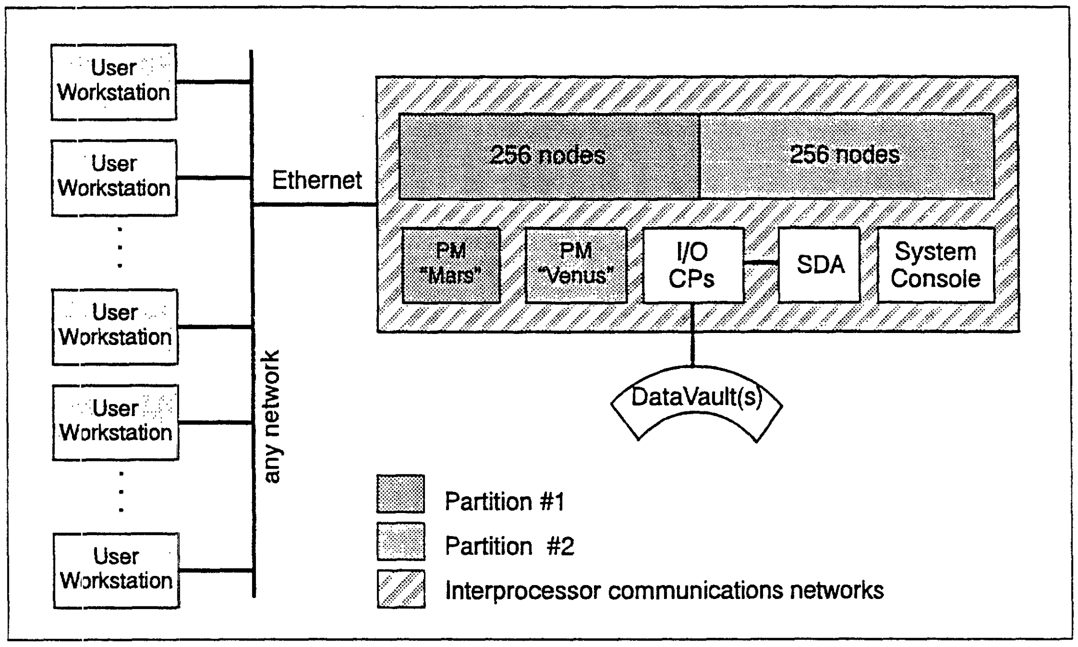

Figure 1. A sample CM-5 system.

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 immutable. 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 standalone 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 its 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 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 binary 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 strict 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.
`cmmd-ld`	The CMMD 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.
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.
`CMMD_error`	The CMMD interface to the `CM_panic` and `CMPN_panic` functions. 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.
`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.
- Chapter 3 describes how to compile and link message-passing programs.
- 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.

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

2.1 Basic Components of a Message-Passing Program

The source code for a CM-5 message-passing program depends on the programming model in use:

- For a hostless program there is a single set of source code files for the nodes. In the hostless programming model, the host merely initiates execution of the node program, and thereafter acts as an I/O server for the nodes. This is the recommended method for writing CMMD programs.
- For a host/node program there are two sets of source code files, one for the host and one for the nodes. The host program must explicitly start and monitor the execution of the node programs.

These two methods for constructing CMMD programs are described in the sections below.

2.2 Hostless Programs

Hostless programming is the recommended method for using CMMD in a message-passing program. In hostless mode, the user writes a single node program, which will run on all the nodes. This program does all computation and communication; it does not communicate explicitly with the host.

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 synchronous message, or to participate in a global instruction, to do I/O, etc.).

What runs on the host, in this mode, is an internal server program provided by the CMMD library itself. This program

- Enables CMMD.
- Downloads the user code to the nodes, which then begin executing it.
- Goes into a polling loop as an I/O server, so that it can communicate with I/O devices on behalf of the nodes. This allows node programs to do I/O.

Hostless CMMD programming is supported for the following languages: C, C++, Fortran 77, CM Fortran, and C*.

2.2.1 Computation and Communication

The code that runs on the nodes performs all the normal tasks of an application program. Computation on each node is written normally.

Communication among nodes uses CMMD function calls. Communication can be point-to-point, when one node sends a message to a second node; or it can be global, with all nodes contributing to the message (and, usually, with all nodes receiving the result). Global communications synchronize all the nodes; point-to-point communications can be either synchronous or asynchronous.

2.2.2 I/O

CMMD provides four modes of I/O:

- *local independent* — each node opens and accesses files directly
- *global independent* — a single file is opened on all nodes simultaneously, with each node having its own, independently-moveable file position indicator.
- *global synchronous broadcast* — a global file is accessed by all nodes simultaneously, with each node reading (or writing) the same data
- *global synchronous sequential* — a global file is accessed by all nodes simultaneously, with each node reading (or writing) its own portion of the file.

A file or stream opened in any one of the three global modes can be changed to one of the other global modes at any time; in fact, a single node can independently change its own view of the mode of a file to facilitiate its own purposes, without affecting the mode of the file as seen by other nodes.

The global modes allow all nodes to open and access a file simultaneously, using only a single UNIX file descriptor. (In local independent mode each node can potentially access a different file, and a separate file descriptor is needed for each file opened by each node.) Because UNIX sets a limit on the number of files that can be open at any one time, the global file modes provide a significant increase in flexibility, allowing the nodes to access multiple files simultaneously without the threat of running out of file descriptors.

The usual UNIX streams — **`stdin`**, **`stdout`**, and **`stderr`** — are available to the nodes. Since a CMMD program is a single process, all nodes share the same streams. Therefore, they must take care not to overwrite each other's contributions to the streams. This is usually accomplished by putting the streams into append mode; see the *CMMD Reference Manual* for details.

2.2.3 Termination

To terminate a hostless program normally, all nodes should call **`exit()`**. Execution of this call on all nodes releases the host from the I/O server and allows the host program to exit normally. Note that **`exit()`** is called automatically when the **`main`** routine returns.

2.3 Host/Node Programs

Host/node programs have a user-written main program that runs on the host (that is, on the partition manager).

In this model, the host program starts parallel operations by calling `CMMD_enable()`. This function:

- gathers `argv`, `argc`, environment variables, and process data in a process control block.
- downloads node code, and thus allows the nodes to start running.
- downloads the process control block.
- starts up the CMMD I/O server, which goes into a polling loop, waiting for I/O requests from the nodes. The server stops only when (1) it is explicitly disabled, or (2) the program terminates abnormally.

NOTE

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

The node program must have all nodes call `CMMD_enable_host()`. This function disables the host I/O server, and allows the host to return from the `CMMD_enable` call and execute the rest of the user-written host program.

In order to perform I/O, the node program will have to re-enable the I/O server. CMMD provides server functions for this purpose, as explained in the chapter on host/node programming in the *CMMD Reference Manual.*

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.

Host programs, when used, generally perform computations, make CMMD calls to communicate with the nodes (perhaps to provide input or receive output from them), and make calls to other libraries or routines.

The CMMD library provides specific functions for communication between host and nodes; these are discussed in the *CMMD Reference Manual*, in the chapter on host/node programming.

Host/node programs terminate only when the host program calls **exit()**. Having the node program call **exit()** merely puts the nodes into a busy-wait state in the OS dispatch loop.

Host/node CMMD programming is supported for these languages: C, Fortran 77, and CM Fortran. Note: Host/node programming is not yet supported for C*, and for CM Fortran programs the host program cannot be written in CM Fortran.

2.4 A Few Caveats

1. If your program hangs, a node is probably waiting for a message that has not been sent.

2. 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.

3. 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 currently available on each node.

4. In host/node programs, the host is responsible for periodically polling the network for I/O client requests. Neglecting to do this is a frequent source of problems. (See the *CMMD Reference Manual* for more information.)

5. For the reasons mentioned above, we recommend that before formulating any program in a host/node style, you consider whether it is possible to write your program in a hostless style. This avoids many of the potential problems involved in host/node programming.

2.5 Fortran I/O Issues

CMMD I/O is based on UNIX I/O. UNIX expects file I/O to be unbuffered, and expects user applications to have a relatively high degree of control over the timing of opens, reads, writes, etc. Fortran I/O, on the other hand, is based on buffered I/O. This causes certain problems for synchronous I/O, as explained below.

2.5.1 CMMD I/O Mode Restrictions

Buffered I/O, by its nature, removes the user's control over when the underlying read or write occurs. For instance, if `stdio` is used to write to a file, the actual UNIX write call will occur when the total number of bytes written to the file reaches the buffer size (typically 8K), rather than at each call to a `stdio` library routine. This means that it is difficult or impossible for the programmer to guarantee synchronous calls across the partition to the underlying I/O routines. Situations in which contributions to a write from some nodes fit into the I/O buffer, while contributions from other nodes overfill the buffer, and thus require multiple writes, are particularly troublesome.

Synchronous I/O Modes

This restriction results in a difficulty for Fortran I/O under synchronous sequential and synchronous broadcast modes, as the only provided Fortran I/O mechanisms are built on top of a layer of buffering. (Synchronous broadcast mode may work correctly, as long as the number of bytes read or written, and any line buffering done, is identical on each node.)

Two workarounds are possible for I/O in synchronous modes:

- Fortran 77 programs may be able to modify their I/O buffer sizes so that an entire read or write for each node fits into the buffer. (Use the Sun `fileopt='BUFFER=`*n*`'` extension to the open statement, where *n* is the desired buffer size.)
- To allow Fortran programmers to access the underlying, unbuffered, UNIX I/O calls, CMMD provides global routines that resolve to these read and write calls.

The global I/O CMMD routines are global read and write calls:

```
INTEGER FUNCTION CMMD_global_read(unit, buffer, length)
INTEGER FUNCTION CMMD_global_write(unit, buffer, length)
 character*(*) buffer
 integer unit, length
```

The `unit` argument is a standard Fortran file unit number; `length` is the number of bytes to be read or written. On success, these functions return the number of bytes written or read; they return -1 on error. A program can use standard Fortran I/O functions to write data into character buffers, and then call the global CMMD I/O functions with the character data buffers as arguments.

Programmers should note the following issues regarding these functions:

- These functions take unit numbers as arguments. Any flags (for example, `status=new`) that are checked at the time the units are opened are still checked. Any other flags that affect writing to the unit (such as record format, etc.) are ignored.
- The global I/O functions can be called only on file units in synchronous-sequential or synchronous-broadcast mode. They will return errors if called on file units that are in local or in global independent mode.
- You can access any given file unit either by Fortran `read` and `write` statements, or by these CMMD functions, but not by both. (Using both may work, but this usage is not supported.)
- The files created by `CMMD_global_write` will be normal UNIX files, not standard Fortran files; in particular, they won't have record separators. (This may or may not be a problem for a given application.)

Global Independent I/O Mode

There also can be problems using global independent mode from Fortran 77 and CM Fortran.

In Fortran 77 you can handle any buffering problems by explicitly using the `fileopt='BUFFER=`*n*`'` option. (While this may not always be necessary, it is good practice to set this flag explicity, nevertheless.)

In CM Fortran the proper method to avoid potential problems is to state explicitly the number of bytes being read or written in the read and write commands. (In other words, don't use the "*" specification.)

Local I/O Mode

I/O in local mode is not affected by these buffering problems.

2.5.2 Other Fortran I/O Considerations

Initializing Standard I/O

CM Fortran initializes standard I/O on demand, rather than at the beginning of a program. Thus, the user cannot change the I/O mode of the standard units until they are used. An application should, therefore, use a standard unit once with its default mode; then change it to the desired mode.

The default mode for the standard I/O ports is `sync_bc_mode`; `independent` mode is generally preferred for `stdout` and `stderr`.

Note: This problem will be fixed in a forthcoming release of CMF.

Using the -f Option in Fortran 77

It is possible for a Fortran 77 program to contain double precision data which is not doubleword aligned in memory (Fortran 77 does not force double alignment by default). The Fortran 77 versions of certain CMMD routines will get a bus error and crash the program if they are called on such data. (This is because CMMD 3.0 routines always attempt double loads when dereferencing double pointers.) To avoid this problem, compile with the -f option.

Don't Mix F77 I/O and CMF I/O

If your main program is in Fortran 77, you should use Fortran 77 for all I/O; if your main program is in CM Fortran, you should use CM Fortran I/O throughout. Do not try to mix Fortran 77 I/O with CM Fortran I/O.

Chapter 3

Compiling and Linking CMMD Programs

In CMMD Version 3.0, there are two kinds of programs:

- hostless programs
- host/node programs

The compiling and linking steps for these two program types are described in detail in the sections below. (Note that compilation of CM Fortran and C* programs is described in a separate section below.)

3.1 Include Files

For both hostless and host/node programs, the include files required by CMMD are the same:

For C-based programs (that is, programs written in C, C++, and C*), the include file to use is:

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

For Fortran-based programs (programs written in Fortran 77 and CM Fortran), the include file is:

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

3.2 Hostless Programs

In hostless mode, the user writes a single C, C++, or Fortran 77 program, which will run on all the nodes. The program does all computation and communication on the nodes; it does not communicate explicitly with the host.

Note: See Section 3.4 for special information about compiling CMMD programs written in CM Fortran or C*.

What runs on the host, in this mode, is a server program provided by the CMMD library itself. This program

- Enables CMMD.
- Downloads user code to the nodes, which immediately start processing it.
- Goes into a polling loop as an I/O server, so it can communicate with I/O devices on behalf of the nodes. This allows user programs to do node I/O.

To terminate a hostless program normally, all nodes should call `exit()`. Execution of this call on all nodes releases the host from the I/O server and allows the host program to exit normally. Note that `exit()` is called automatically when the `main` routine returns.

3.2.1 Compiling and Linking Hostless Programs

Compile as Usual

You compile a hostless program just as you would an ordinary C, C++, or Fortran 77 program. You use whatever compiler (Sun or GNU) you would normally use, and you supply your own makefile. The following compiler versions are supported:

- Sun F77 versions 1.* are supported. Version 2.0 will compile and run, but is not supported by the pndbx debugger.
- All Sun bundled C compilers are supported. Versions of acc (unbundled Sun C) prior to version 2.0 are supported. Version 2.0 will compile and run, but is not supported by the pndbx debugger.
- All versions of GNU C up to and including version 2.3.3 are supported. The Sun CFront (C++) compiler version 1.0 is supported. The GNU G++ (C++) compiler version 2.3.3 and earlier is supported.

Link with cmmd–ld

After compiling your program, you must link it using the customized linker, `cmmd-ld`. Some options are:

`-comp` *comp*	Required. Specifies the command to be used for linking (`cmf`, `gcc`, `f77`, etc.)
`-comphost` *comp*	Optional. Selects different command to be used for host files in host/node linking (see Section 3.4 below).
`-vu`	Optional for CM-5 systems with vector units. Indicates that code should be linked with a VU-compatible library.
`-node` *object_code.o library.a ...*	Required. Specifies the object code and library files that you want linked for the node programs.
`-host` *object_code.o library.a ...*	Optional. Specifies alternative object code and library files that you want linked for the host program.
`-o` *executable–name*	Optional. Specifies executable name (default `a.out`).
`-g`	Optional. Links in the CMMD debugging libraries.
`-v`	Optional. Requests verbose descriptions of the linking process.
`-l`, `-L`, etc.	Optional. Pathnames of any required user libraries. (Same as corresponding argument of Sun `ld`.)
`-cmos_root` *path*	Optional. Tells the linker where to find the CMOS system software, if not in `/usr/lib`.
`-cmmd_root` *path*	Optional. Tells the linker where to find the CMMD library, if not in `/usr/lib`.

`cmmd-ld` first uses the specified compiler to link the program in the normal manner; it then customizes the resulting file for execution with CMMD, adding its own host program. An on-line manual page is available for `cmmd-ld`.

Usage Note: If you attempt to compile a program with many source files or very long pathnames, you may run into `cmmd-ld`'s command line length limit, which is currently 4K characters. A simple workaround for this limit is to collapse a number of the object files you are linking by using `ld`:

```
ld -r -o <output>.o <list of input .o's>
```

3.2.2 Important Note for VU CM-5 Sites:

If your CM-5 has Vector Units, you can use the `-vu` switch to cause `cmmd-ld` to link with the VU version of the software, and thereby make use of the VUs. **Note:** You *must* supply this switch when linking a CMMD program written in a parallel language like CM Fortran or C*.

3.3 Host/Node Programs

You compile and link host/node programs in almost the same way as hostless programs. For host/node programs, however,

- You must specify your own host program.
- When you compile the host program, you must specify `-DCP_CODE`. This allows the use of appropriate include files for the control processor, rather than those for the nodes.
- When linking, you must specify the `-host` option as well as the `-node` option. (The `-host` option specifies the host objects to be linked.)

Note: See Section 3.4 for special information about compiling CMMD programs written in CM Fortran or C*.

3.3.1 Executing Host/Node Programs

Starting the Programs

Host programs must start parallel operation by calling `CMMD_enable()`. The node program cannot run until the host program has made this call.

Node programs must similarly call `CMMD_enable_host()`; until they make this global call, the host program cannot progress beyond its own `CMMD_enable()` call. The reasons for this concern the CMMD I/O server, and are explained in the section on the server, below.

The I/O Server

When a host program starts running on the partition manager, it must call `CMMD_enable`.

`CMMD_enable`, when called, does the following:

- It gathers `argv`, `argc`, environment variables, and process data in an process control block.
- It downloads node code, and thus allows the nodes to start running.
- It downloads the process control block.
- It starts up the CMMD I/O server, which goes into a polling loop, waiting for I/O requests from the nodes. The server stops only when (1) it is explicitly disabled, or (2) the program terminates abnormally.

While the server is running the host processor cannot do anything else; in particular, it cannot run any code in your host program.

`CMMD_enable_host()`, when issued by each of the processing nodes, turns off the server, and thus allows user-written host code to execute.

If the node program needs to do I/O, it must either supply its own I/O server program, or re-enable the CMMD server program. Use the server routines described in the *CMMD Reference Manual* to do this.

Terminating the Programs

A host/node program terminates normally when the host program calls `exit()`. A call to `exit()` made on a node merely terminates the node program.

3.4 Compiling CM Fortran and C* Programs

To compile and link a hostless CMMD program written in CM Fortran (that is, to use CM Fortran "on-a-node"), use the CM Fortran compiler (`cmf`) as usual, and specify the **`-node`** switch to indicate on-a-node CM Fortran execution. (When compiling on a CM with vector units, you *must* also supply the **`-vu`** switch to insure proper linking of the program.)

To compile and link a host/node CMMD program written in CM Fortran, you can use the parallel compiler (`cmf`) to compile and link the host and node programs by preceding each host source file with the **`-host`** switch, and supplying the **`-comphost`** switch to specify the compiler to be used for the host program. (**Important:** The host program currently cannot be written in CM Fortran, and the **`-comphost`** argument cannot be the `cmf` compiler.)

To compile and link a hostless CMMD program written in C*, use the C* compiler (`cs`) as usual, and specify the **`-node`** switch to indicate on-a-node execution. (When compiling on a CM with vector units, you *must* also supply the **`-vu`** switch to insure proper linking of the program.)

Note: In a hostless C* program, the node program's main routine *must* be written and compiled in C* so that C*-specific initializations are included.

The host/node programming model is currently not supported for C*.

3.5 Fortran 77 Programs

3.5.1 Use –Nx Option to Increase Symbol Table Limit

Fortran 77 programmers may find that the default Fortran 77 symbol table size is not large enough to contain the symbols used in a CMMD program. You can tell this is the case if you see the following error when you try to compile a program:

```
f77 program.pn.F ...
Compiler error: Too many external symbols.
```

A simple workaround is to specify the **`-Nx`** option to increase the size of the symbol table:

```
f77 program.pn.F ... -Nx500
```

3.5.2 Use –f Option to Align Double Precision Data Correctly

It is possible for an Fortran 77 program to contain double precision data which is not double aligned (Fortran 77 does not force double alignment by default). The Fortran 77 versions of certain CMMD routines will get a bus error and crash the program if they are called on such data. (This is because CMMD 3.0 routines always attempt double loads when dereferencing double pointers.) To avoid this problem, compile with the `-f` (or `-dalign`) options.

3.6 Writing Version-Independent CMMD Code

3.6.1 Important Differences Between Versions 2.0 and 3.0

CMMD 3.0 programs must not include the header file `cmmd-io.h` (a compile-time error is signalled if this is done). The `cmmd-io.h` file is automatically included by `cmmd.h`.

The Version 2.0 error function `cmmd_error` is called `CMMD_error` in Version 3.0.

Functions that have handlers (eg. `CMMD_send_async`) are extended in 3.0 to include an extra argument that is passed to the handler function when it is called. This extra argument is specified as `void *`, and thus can be a pointer to anything that the programmer wishes.

Handler functions in C are passed a pointer to a `CMMD_mcb`, thus requiring an extra level of indirection than was required in Version 2.0. This is in contrast to Fortran handler functions, which are passed a `CMMD_mcb` instead. The reason for this difference is Fortran's call-by-reference strategy, which implicitly includes the extra level of indirection. By providing the same extra indirection in C programs, CMMD can use the same internal interface for both languages.

3.6.2 CMMD_VERSION Preprocessor Symbol

There is a C preprocessor symbol, `CMMD_VERSION`, that you can use to make your code compatible with both Version 2.0 and Version 3.0 of CMMD.

Version 3.0 defines it as: `#define CMMD_VERSION 30`

Version 2.0 defines it as: `#define CMMD_VERSION 20`

This allows you to include constructs like the following in your code:

```
#if CMMD_VERSION == 20
#include <cm/cmmd-io.h>
#endif
```

3.7 Sample Programs

A number of sample CMMD programs are included with this release, and can be found in the directory:

`/usr/examples/cmmd/{hostless,hostnode}/`*language*`/`*example*

3.7.1 Hostless Examples

C Examples:

am_fetch_ring	am_performance	am_store_ring
channel_perf_parallel	channel_perf_serial	channel_wave
hello	int_parallel_perf	int_serial_perf
io	master	mp_parallel_perf
mp_serial_perf	performance	port_perf_parallel
port_perf_serial	redist_nonblk_hndlr	subset_broadcast

Fortran 77/CM Fortran Examples:

simple	hello
io-workaround	n-body-bc

C++/G++ Examples: `potato`

3.7.2 Host/Node Examples

C Examples:

cmmd_canned_host	global_perf	hello
io_plus	performance	x_io_server

Fortran 77 Example: `simple`

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 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

32 PN System, 7572K memory free, 3 Processes, Daemon up: 15:15

USER   PID    CMPID  TIME   TEXT+DATA  STACK  PSTACK  PHEAP   COMMAND

sal    4722    1     14:51  6216K       48K     64K    4K     radix
kim    4949    2      6:39  1428K       68K      0K    0K     get.hw
jan   *6111    3      3:14   288K       48K   2432K    0K     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 refer only to the nodes. The *stack* is the UNIX process stack on each node, while *pstack* and *pheap* refer to parallel 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 can have up to 64 timers active (identified by integers from 0 - 63). Each node's timers record times only for that node.

Note that these timers execute on the nodes only. CMMD no longer offers timers that execute on the host.

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 operating system's dispatch loop. By design, this does not occur in CMMD.
- *Elapsed time* is the sum of busy time and idle time. It represents the amount of time during which the process was scheduled for execution on the CM-5. Thus, it measures time only during this program's timeslices.

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

Note: The CMMD timers do not measure wall-clock time. The UNIX functions `gettimeofday()` and `ctime()`, callable on the node (or on the host), can provide this measure.

The CMMD timing functions are

> int **`CMMD_node_timer_clear`** (*int timer*)
> int **`CMMD_node_timer_start`** (*int timer*)
> int **`CMMD_node_timer_stop`** (*int timer*)
>
> double **`CMMD_node_timer_busy`** (*int timer*)
> double **`CMMD_node_timer_elapsed`** (*int timer*)
> double **`CMMD_node_timer_idle`** (*int timer*)

For more information on using timers, see the *CMMD Reference Manual*, or the on-line manual pages.

4.6.1 Using CMOST Timers

It is also legal to use CMOST timing routines (**`CM_timer_start`**, **`CM_timer_clear`**, **`CM_timer_stop`**) in your CMMD code — the two sets of timers are separate and compatible, and operate similarly. When called from CM Fortran, for example, the CMOST functions gather timing information independently on each node. You must, however, remember to include the appropriate CMOST header files for the timers you use. For example, the CM Fortran interface to the CMOST timers is given by the header file

```
#include <cm/timer-fort.h>
```

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`, and their CMMD interface, `CMMD_error`.

5.1 Error Handling

The Connection Machine operating system provides two error handlers:

`CM_panic` (*"error_message"*)	for host programs
`CMPN_panic` (*"error_message"*)	for node programs

CMMD provides an equivalent function,

```
int CMMD_error("error_format", [args] ...)
```

The syntax of this function is the same as the syntax of the UNIX function `printf`. The "`error_format`" argument can contain a character string (that is, an error message) that is copied into the output stream; it can also contain conversion specification for the optional args. (See the on-line man page for `printf` for details regarding args.) If successful, `CMMD_error` does not return.

The `CMMD_error` function can be used on either host or node programs; it calls the appropriate OS function in either case. Users are encouraged to use `CMMD_error`, rather than the OS functions. It provides the cleanest method for terminating a program in case of error.

Both the OS functions and the CMMD function allow you to supply your own error message. 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

Having any node call the exit routine (`EXIT` or `STOP` in Fortran, `exit()` in C) does not terminate the program; it merely halts that particular node, which then waits for all other nodes to exit.

5.1.1 Default Error Handling

The default behavior for the panic 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` (and therefore of `CMMD_error`) in a number of 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 experienced UNIX programmers (wizards, in other words) should use these methods, and regardless of level of experience should consult the manual pages for `CMOS_abort` and `CM_longjmp`.

5.2 CMMD Safety Routine

CMMD 3.0 includes a pair of routines that can be used to enable and disable safety-checking in your program:

```
CMMD_enable_safety()
CMMD_disable_safety()
```

These two functions enable and disable CMMD safety mode. When invoked, they set a global variable and return immediately, with no return value. When safety mode is enabled, CMMD send and receive functions, channel functions, and associated utility routines signal errors if the user either passes recognizably illegal arguments or calls the functions under illegal circumstances. For a list of the errors signalled by safety checking, see the CMMD *Reference Manual.*

5.3 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, plus the host `core` file, if any, should help you to track down the cause of the error.

5.3.1 Using printf

Any node can call the `printf` routine to print out data. The output is printed to `stdout`. It is important that the user set the I/O mode for `stdout` and `stderr` (unit 6 and unit 1 in Fortran 77) to `independent` while debugging. In fact, it is always a good idea to have `stderr` set to independent mode.

Note that this behavior differs from programs run in back-compatibility mode. For such programs, output from `printf` is stored into the file `CMTSD_printf.pn.`*pid* in the current directory (where *pid* is the process ID of your program). Using this form of `printf` will slow down your program a great deal, so it is best used only for debugging purposes.

Warning: The `printf` routine, like all I/O functions, does polling, so if your CMMD program depends on explicit polling for its operation, incautiously inserting `printf` statements may have unexpected effects.

5.3.2 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 the error was caused by a segmentation fault, and so on.

5.3.3 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.

To save disk space, only unique cores are dumped. Thus, if several nodes have the same error, only the core for the first node with that type of error is dumped. The first node with no error (if there is such a node) will also dump core.

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

You can look at PN core files with **`pndbx`**; see Section 6.9.4.

5.3.4 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.

5.4 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 reliable. Remember that the host and the nodes are not necessarily synchronized. 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 using the host/node model on the CM-5, you are actually debugging two programs simultaneously but separately. Even when you use the hostless model, you must still invoke a debugger on the CMMD-supplied main program in order to get the node program started. There are two methods for doing the debugging:

- You can debug your host program inside the Prism programming environment. You use Prism's own windowed debugger to debug the host program (or the CMMD-supplied main program) and **pndbx** (invoked from within Prism) to debug the node program. You can use Prism for both C and Fortran main programs; you cannot use it for C++ programs.

- You can use **gdb** or the standard UNIX debugger **dbx** to debug the host program, and the **pndbx** debugger to debug the node program.

  Note: **dbx** and **gdb** have problems with some internal symbols of the CMMD debugging (**-g**) library — you may not be able to use these two debuggers with the debugging library.

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.

This chapter presents an 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 6.11 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 (see Section 6.3).

`stop in` *procedure*	Sets a breakpoint at start of procedure.
`stop at` *line*	Sets a breakpoint on the specified source line.
`cont`	Continues after being stopped by a breakpoint.
`step, next`	Single-steps into or over subroutines.
`print` *exp*	Prints value of variable or an expression.
`assign` *var* = *exp*	Assigns a value to a variable.
`where`	Provides a stack trace.
`file` *name*	Changes the current source file.
`use` *directory–list*	Sets list of source file search directories.

6.2.2 Other Commands

Features of `dbx` that involve querying the symbol table or source file, such as **`func`**, **`list`**, and **`whatis`**, 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*	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 `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`", "`where running`", "`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 - $f31`	
`$psr`	processor status register
`$pc`	program counter
`$npc`	next program counter

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 `$l7`.
`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`, it is important to compile the source code for your host program (if you are using one) and your node program with `-g`.

Important: Don't use the `-g` option when linking your program! Linking with the `-g` option causes your program to link with the debug versions of the CMMD libraries; this will cause the debugger to try to find the CMMD library sources. The debug versions of the libraries are useful only for debugging CMMD sources; they are of no use to user applications.

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

For complete information on Prism itself and how you use it, see the *Prism User's Guide.*

To use `pndbx` within Prism, follow these steps:

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

   ```
   % prism master
   ```

2. A Prism window appears on your screen, displaying the specified host program or the CMMD-supplied main program. Set a breakpoint in the program just after initialization by issuing this command on the Prism command line:

   ```
   stop in cmmd_debug
   ```

3. Run to this breakpoint by clicking on **Run**, or by issuing the `run` command with the appropriate command-line arguments.

4. Click on the **Utilities** menu and choose **PN Debug**. A new window opens containing `pndbx`, already invoked and active on your node program.

5. Click on **Cont** in Prism to continue execution of the host program or CMMD-supplied main program.

6.6.2 Using dbx

To debug a program using `dbx` and `pndbx`, follow these steps:

1. In one window, load your host program (if any) into `dbx`.

2. Issue the command `stop in cmmd_debug` to set a breakpoint just after initialization.

3. Start the host program (either your own or the CMMD-supplied host server program) by issuing the `dbx` command `run`.

4. In a separate window, issue the `cmps` command to obtain the process ID of the stopped process.

5. Invoke `pndbx` in the second window by specifying the name and process ID of the program. For example:

   ```
   pndbx master 14640
   ```

6. Continue execution of the host program with the `dbx` command `cont`.

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; the `pndbx` prompt identifies which node you are monitoring. For example:

```
(pndbx 0)
```

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

The output for `pnstatus all` (and some other commands) by default is not paginated. If you have long output, you may want to have **pndbx** paginate the output by printing a `more?` prompt after a specified number of lines. You can change the pagination with the **pndbx** command

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

Setting page size to 0 (the default) 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.

IMPORTANT!

If you interrupt a node while it is handling a message, and then call a routine that uses the CM-5 networks, you may crash the timesharing daemon.

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.

6.9.4 Using pndbx to Debug Core Files

See Section 5.3.3 for a discussion of core files. You can invoke **pndbx** on a PN core file by specifying its name on the command line, after the name of the executable program with which it is associated. For example:

```
% pndbx myprog CMTSD_core.pn1.1906
Welcome to Pndbx version 1.1
Type 'help' for help.
Current partition size is 64
reading core file ...
reading symbolic information ...
(pndbx 0)
```

When you use **pndbx** on a core file, you are restricted to examining the state of the single node that was captured in the core file; you cannot switch nodes via the **pn** command. Also, as with **dbx**, these restrictions apply:

- You cannot issue execution commands (**run**, **cont**, etc.).
- You cannot issue commands that write to memory (**assign**, **stop in**, etc.).

Apart from these restrictions, full debugging capabilities are available. In particular, you can examine the state of the node with commands like **where**, **print**, and so on.

6.9.5 Debugging CM Fortran Programs with pndbx

You can use **pndbx** to debug CM Fortran programs. To do this, compile and link the program with the debug "**-g**" flag, to ensure that the proper debugging information is included and that the debugging version of the CMMD library is linked in. (**Note:** The debug flag disables some compiler optimizations.)

You can then debug your program as usual using **pndbx**. The **pndbx** debugger understands all standard CM Fortran data types (**integer**, **real**, **double**, **complex**, and **double complex**). Arrays are printed in their entirety, one element per line. (The built-in variable **$print_width** can be used to change this default). Array sections can be specified using CM Fortran syntax. Arbitrary expressions can be evaluated, with some restrictions. The "assign" command can be used to modify variables.

Users who need to get at the lower level details of CMF array descriptors can use the construct *&*array* to view the contents of the array descriptor itself.

The following example illustrates these features:

```
(pndbx 0) whatis u
(CM based) double precision U(1:6)
(pndbx 0) print u
(1)       1.1
(2)       1.1
(3)       1.1
(4)       1.1
(5)       1.1
(6)       1.1
(pndbx 0) print u(1:4)
(1)       1.1
(2)       1.1
(3)       1.1
(4)       1.1
(pndbx 0) set $print_width = 2
(pndbx 0) print u
(1:2)    1.1       1.1
(3:4)    1.1       1.1
(5:6)    1.1       1.1
(pndbx 0) print u(1:4)+1
(1:2)    2.1       2.1
(3:4)    2.1       2.1
(pndbx 0) assign u = 2.2
(pndbx 0) print u
(1:2)    2.2       2.2
(3:4)    2.2       2.2
(5:6)    2.2       2.2
```

```
(pndbx 0) print &u
CM array, descriptor address = 0xb8aa4 (print *&U to see
the entire descriptor)
(pndbx 0) print *&u
(desc_or_object_kind = 1025, debug_info_ptr = 0xb8a98,
 element_type = 5, sparel = 0, spare2 = 0,
 cm_location = 1342187272, user_rank = 1,
 spare4 = 757192, spare 5 = 757084, home = 3,
 initial_data = -1, is_modified = 0,
 array_geometry = 1468752, spare6 = -1, spare7 = 1,
 spare8 = 757080, spare9 = -1, is_slicewise = 1,
 element_size = 8)
```

6.10 Submitting a Bug Report

If you're experiencing a persistant, inexplicable problem with a program, and wish to send a bug report to Thinking Machines Corporation Customer Support for assistance, here is the information you should include:

- Whether the program worked correctly with previous CMMD releases, and if so, which version of the software you were using then.
- The version of the CMMD software you are using now.
- If the program is small enough, a copy of it and instructions for compiling and running it. (Alternatively, if you can reduce the problem to a small section of code that runs on its own, provide that instead.)
- If your program produces a **CMTSD_errors** output file, cut and paste the text for each node that signaled an error. The node error reports should look like this:

```
******* Summary from pe N *********

------ PN Status -------
Program counter = 0x43e98
Stack pointer = 0xf7fe3648
Processor status = 0x11401081
Floating point status = 0x60000
```

```
-------- PN Faults --------
Error detected in this PN...
```
(more information follows)

- If your program produces any **CMTSD_core.pn*nnn*** output files, for each one of these files, do the following:

  ```
  % pndbx your-program CMTSD_core.pnnnn.pid
  where
  $g0/32X
  print $pc
  <xxxxyyyy> <--- retype this number below:
  print <xxxxyyyy>-32
  <zzzzzzzz> <--- retype this number below:
  <zzzzzzzz>/10i
  <a series of instructions will be printed out>
  quit
  ```

  And include the information you see printed out by **pndbx**.

- Finally, if your program uses the VUs, include the contents of the **CMTSD_dp.pn*nnn*.pid** file, for each node that signalled an error. (**Note:** This should follow any other information you have collected above.)

By gathering this information *before* you report a problem, you'll make it much easier for Customer Support to expedite a solution to the problem.

6.11 A Sample pndbx Session

This session shows how to use some **pndbx** commands in debugging a hostless CMMD program called **master**, which approximates pi using a Monte Carlo method. Assume two windows, one for running **dbx** to start the CMMD-provided main program, and one for debugging the nodes with **pndbx**.

We begin by starting up **dbx** on the PM and executing to the start of **cmmd_debug**. (The arguments to **run** specify first, the number of trials, and second, the number of trials per work period.)

dbx Window

```
bowker% dbx master
Reading symbolic information...
Read 19967 symbols
(dbx) stop in cmmd_debug
(2) stop in cmmd_debug
(dbx) run 1000 250
Running: master 1000 250
stopped in cmmd_debug at 0x3208
cmmd_debug:      save    %sp, -80, %sp
Current function is main
   33      CMMD_enable();
```

We can then find out the process ID:

pndbx Window

```
bowker% cmps
4 PN System, 13356K memory free, 2 Processes, Daemon up:  1:35

USER        PID CMPID     TIME TEXT+DATA  STACK  PSTACK    PHEAP  COMMAND
philip     2273    0      5:28     1268K    48K      0K       4K  t
bowker   * 2443    1      0:09      608K    48K      0K       0K  master
```

And start up **pndbx**:

pndbx Window

```
bowker% pndbx master 2443
Welcome to Pndbx version 1.1 of 6/11/92 10:38 (cherry-gar-
cia.think.com).
Type 'help' for help.
Current partition size is 4
reading symbolic information ...
```

By default we are monitoring node 0, which in this program is the master node. We do a status and list some source code:

pndbx Window

```
(pndbx 0) pnstatus all
PN  pn-status
 0  running
 1  running
 2  running
 3  running
(pndbx 0) list master
interrupt in CMNA_bc_wait_for_receive_ok at 0x279c4
CMNA_bc_wait_for_receive_ok+0xc:
andcc  %o0, 16, %g0
   53    work_increment; /* increment of work done by each PE
*/
   54    {
   55
   56        int
   57          work_left,  /* loop variable for number of
trials left to perform */
   58          numhits = 0,    /* number of hits from Monte
Carlo trials */
   59          numworking = 0, /* number of workers working
        */
   60          answer;         /* buffer for answers returned
from workers */
   61
   62
   63        char
```

We then set a breakpoint. Note how as a side effect of setting a breakpoint, the node goes into a "break" state.

pndbx Window

```
(pndbx 0) cont
(pndbx 0) stop at 85
[1] stop at "/users/cmsg3/bowker/cmmd/mas-
ter.c":85
(pndbx 0) pnstatus
  break
```

Since the CMMD-provided main program is stopped, we must continue it to allow the nodes to proceed:

dbx Window

```
(dbx) cont
```

The main program will continue to run until the process exits. To continue execution, we do a `cont` on node 0, which runs the master process:

pndbx Window

```
(pndbx 0) cont
```

This causes the following output to be displayed in the **dbx** window:

dbx Window

```
Master/worker using 4 pn's to perform 1000 trials
        Work increment is 250
```

Now, going back to the node window, we wait for node 0 to reach its breakpoint at line 85, then display its call stack:

pndbx Window

```
(pndbx 0) wait
[1] stopped in .master.master at line 85 in file "/users/
cmsg3/bowker/cmmd/master.c"
(pndbx 0) where
   85              if (work_left < work_increment) {
.master.master(numtrials = 1000, work_increment = 250),
line 85 in "/users/cmsg3/bowker/cmmd/master.c"
main(argc = 3, argv = 0xf7fff60c), line 242 in "/users/
cmsg3/bowker/cmmd/master.c"
```

We print the value of a variable that shows the number of trials left, then continue and print the value after node 0 hits its breakpoint again:

pndbx Window

```
(pndbx 0) print work_left
1000
(pndbx 0) contw
[2] stopped in .master.master at line 85 in file "/users/
cmsg3/bowker/cmmd/master.c"
 85         if (work_left < work_increment) {
(pndbx 0) print work_left
750
```

We can also look at what's happening in other nodes:

pndbx Window

```
(pndbx 0) pn 1
(pndbx 1) where
interrupt in CMMD_receive_block(source_node = 0, tag =
-2147483648, buffer = 0xf7fff53c, bytes = 4), line 1781 in
"message.sh.c"
CMMD_receive(source_node = 0, tag = -2147483648, buffer =
0xf7fff53c, bytes = 4), line 1767 in "message.sh.c"
worker(), line 181 in "/users/cmsg3/bowker/cmmd/master.c"
main(argc = 3, argv = 0xf7fff60c), line 244 in "/users/
cmsg3/bowker/cmmd/master.c"
(pndbx 1) pn 2
(pndbx 2) where
interrupt in CMMD_send_block at line 1655 in file "/users/
cmsg3/bowker/cmmd/master.c"
"/users/cmsg3/bowker/cmmd/master.c" has only 253 lines
CMMD_send_block(dest_node = 0, tag = 2, buffer =
0xf7fff530, bytes = 4), line 1655 in "message.sh.c"
CMMD_send(dest_node = 0, tag = 2, buffer = 0xf7fff530,
bytes = 4), line 1640 in "message.sh.c"
worker(), line 198 in "/users/cmsg3/bowker/cmmd/master.c"
main(argc = 3, argv = 0xf7fff60c), line 244 in "/users/
cmsg3/bowker/cmmd/master.c"0
```

A **cont all** continues all the nodes until node 0's breakpoint is reached again:

pndbx Window

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

We can continue node 0 twice more to complete the specified number of trials and end execution of the program:

pndbx Window

```
 (pndbx 2) pn 0
(pndbx 0) cont
(pndbx 0) cont
```

The results appear in the **dbx** window:

dbx Window

```
        Worker 2 reports 203 hits
        Worker 1 reports 191 hits
        Worker 3 reports 198 hits
        Worker 2 reports 202 hits
        pi = 3.176000
task done

execution completed, exit code is 3
program exited with 3
```

We then exit both debuggers:

dbx Window

```
(dbx) quit
```

pndbx Window

```
(pndbx 0) quit
```

Appendix A

Sample Programs

This appendix contains a number of examples of CMMD programs. These programs, plus their makefiles, are also available on-line.

Examples provided in C are as follows:

- **round_trip:** Round-trip message (host/node).

  The host and all nodes synchronize clocks. Then the host sends a message (zero) to node 0, which receives it and adds one to it; then sends it to node 1, etc., until the message travels around all the nodes and comes back to the host. The host checks if the message is correct (equal to the number of nodes) and prints the result. In addition the host prints the time at which each node received the message.

- **ring:** One-dimensional ring (hostless).

  The nodes communicate in a ring pattern using the synchronous `CMMD_send_and_receive` function. At the end the communications rate is computed and printed.

- **redist1:** Global redistribution using synchronous functions (hostless).

  Node 0 (master) generates *n* = (`num_nodes`-1)*100 random integers in the range (1, `num_nodes`-1) and redistributes them in such a way that a given node receives only the numbers that are equal to its node address. The global asynchronous "or" flag is used to test for completion.

- **redist2:** Global redistribution using asynchronous functions (hostless).

  Each node generates *n* = 100 random integers in the range (0, `num_nodes`-1) and redistributes them in such way that a given node receives only the numbers that are equal to its node address. The global asynchronous "or" flag is used to test for completion.

- **redist3:** Global redistribution using asynchronous functions and a handler (hostless).

  Node 0 (master) generates *n* = (`num_nodes`-1)*100 random integers in the range (1, `num_nodes`-1) and redistributes them in such way that a given node receives only the numbers that are equal to its node address. A handler function counts the number of messages delivered. The global asynchronous "or" flag is used to test for completion.

- **sort:** Enumerate-pack sort (parallel radix) (host/node).

  This sort performs the following operations for each bit of the key: All the elements with a 0 in that bit position are enumerated with a scan with add. Then a second scan does the same for elements with a 1. A scan with max (downward) broadcasts the sum of elements with a 0 to all nodes. This sum is added to the result of the second scan and these values are the new node addresses for elements with a 1 in the bit position. The new node addresses for elements with a 0 is just the enumeration obtained from the first scan. The initial number in each node is randomly generated.

  Two Fortran 77 examples are also provided:

- **simple_io:** A host/node version of a "hello world" program, demonstrating the I/O service loop.

- **redist_block:** Global redistribution using synchronous functions (hostless)

  Node 0 (master) generates *n* = (`num_nodes`-1)*100 random integers in the range (1, `num_nodes`-1) and redistributes them in such way that a given node receives only the numbers that are equal to its node address. The global asynchronous "or" flag is used to test for completion.

A.1 C Examples

A.1.1 round_trip

Round-trip message (host program)

The host and all nodes synchronize clocks. Then the host sends a message (zero) to node 0, which receives it and adds one to it; then sends it to node 1 etc., etc., until the message travels around all the nodes and comes back to the host. The host checks if the message is correct (equal to the number of nodes) and prints the result. In addition each node prints the time at which it received the message.

```
#include <stdio.h>
#include <sys/types.h>
#include <cm/cmmd.h>
#if CMMD_VERSION <= 20
#include <cm/cmmd-io.h>
#endif

main ()
{ int msg_len=4, message, tag=0, i, num_nodes, pid = getpid();
  double times[1024];

  printf("\n Executing host program... (pid=%d)", pid);
  CMMD_enable();
  num_nodes=CMMD_partition_size();

  printf("\n Executing node programs...");
  CMMD_sync_host_with_nodes();    /* Global sync. to synchronize clocks */

  /* send message to node 0, receive message from node num_nodes-1 */

  message = 0;
  CMMD_send(0, tag, &message, msg_len);

  CMMD_receive(num_nodes-1, tag, &message, msg_len);

  /* Read timers */

  CMMD_gather_from_nodes(times, sizeof(double));

  if (message == num_nodes) {
    printf("\n All done ok, message = %d\n", message);
    for (i = 0; i < num_nodes; ++i)
      printf("\n Node %d: msg received ok, time= %f secs", i, times[i]);
  }
  else
    printf("\n Error: number of nodes = message = %d\n", message);

}
```

Round-trip message (node program)

```
#include <stdio.h>
#include <cm/cmmd.h>
#if CMMD_VERSION <= 20
#include <cm/cmmd-io.h>
#endif

main()
{ int src, dest, msg_len=4, message, tag=0, num_nodes, my_address;
  double time;

  CMMD_enable_host();

  /* Compute self address, source and destination */

  my_address = CMMD_self_address();
  num_nodes = CMMD_partition_size();
  src = (my_address == 0) ? CMMD_host_node() : my_address - 1;
  dest = (my_address == num_nodes-1) ? CMMD_host_node() : my_address + 1;

  /* Synchronize clocks */

  CMMD_sync_with_host();
  CMMD_node_timer_clear(0);
  CMMD_node_timer_start(0);

  /* Receive message  */

  CMMD_receive(src, tag, &message, msg_len);

  /* Get time */

  CMMD_node_timer_stop(0);
  time = CMMD_node_timer_elapsed(0);

  /* Add one to the message and send it to the next guy  */

  message++;
  CMMD_send(dest, tag, &message, msg_len);

  /* Host accumulates all times */

  CMMD_concat_elements_to_host(&time, sizeof(double));
}
```

A.1.2 ring

One-dimensional ring

The nodes communicate in a ring pattern using the synchronous **CMMD_send_and_receive** function. At the end the communications rate is computed and printed.

```
#include <stdio.h>
#include <cm/cmmd.h>
#if CMMD_VERSION <= 20
#include <cm/cmmd-io.h>
#endif

main()
{ int src, dest, msg_len=24000, tag=0, num_nodes, i, niter=10, my_address;
  char *msg_in, *msg_out;
  /* Get self address and number of nodes. Allocate msg. buffers */
  my_address = CMMD_self_address();
  num_nodes = CMMD_partition_size();
  msg_in = (char *) malloc(msg_len);
  msg_out = (char *) malloc(msg_len);
  /* set independent IO mode */
  CMMD_fset_io_mode(stdout, CMMD_independent);
  CMMD_fset_io_mode(stderr, CMMD_independent);
  if (my_address == 0) printf("\n Executing ring ...\n");
  /* Compute source and destination */
  src = my_address - 1;
  dest = my_address + 1;
  if (src < 0) src += num_nodes;
  if (dest >= num_nodes) dest -= num_nodes;
  /* Synchronous send and receive loop (repeat niter times) */
  CMMD_node_timer_clear(0);
  CMMD_node_timer_start(0);
  for (i = 0; i < niter; i++) {
    CMMD_send_and_receive(src, tag, msg_in, msg_len,
                          dest, tag, msg_out, msg_len);
    CMMD_sync_with_nodes();
  }
  CMMD_node_timer_stop(0);
  if (my_address == 0)  /* comm. rate = (total bytes in plus out)/time */
    printf("Time=%f [secs], Comm. Rate=%f [Mb/sec/node] \n",
           CMMD_node_timer_busy(0)/niter,
           2.0*niter*msg_len*1.e-6/CMMD_node_timer_busy(0));

}
```

A.1.3 redist_block

Global redistribution using synchronous (blocking) functions

Node 0 (master) generates *n* = (**num_nodes**-1)*100 random integers in the range (1, **num_nodes**-1) and redistributes them in such way that a given node receives only the numbers which are equal to its node address. The global asynchronous "or" flag is used to test for completion.

```
#include <stdio.h>
#include <cm/cmmd.h>
#if CMMD_VERSION <= 20
#include <cm/cmmd-io.h>
#endif

main(){ int msg_len=4, tag=0, i, n, rand_num, bufferin, recv_count=0, total;
  int my_address=CMMD_self_address(), num_nodes=CMMD_partition_size();

  /* set independent IO mode */

  CMMD_fset_io_mode(stdout, CMMD_independent);
  CMMD_fset_io_mode(stderr, CMMD_independent);

  if (my_address == 0) {          /* Node 0: sender node */
    printf("\n Executing redist_block ...\n");

    CMMD_set_global_or(1);        /* set global "or" to 1 (not completion) */
    while(!CMMD_get_global_or()); /* wait for setting */

    CMMD_sync_with_nodes();       /* ready to start */

    srand(1234);                  /* initialize RNG */

    n = (num_nodes-1)*100;        /* each node will get 100 msgs on average */

    for (i = 0; i < n; i++) {     /* send n random numbers to nodes */

     rand_num = 1 + (rand() % (num_nodes-1)); /* generate random number */

      CMMD_send(rand_num, tag, &rand_num, msg_len);
    }

    CMMD_set_global_or(0);        /* set global "or" to 0 (completion) */
    printf("\n Total msgs generated (node 0)= %8d\n", n);
    total = CMMD_reduce_int(recv_count, CMMD_combiner_add);
    printf("\n Total msgs received by nodes (1-- num_nodes-1)= %8d\n", total);

  }
  else {                          /* Nodes (1 -- num_nodes-1): recipients */

    CMMD_set_global_or(0);        /* node 0 will control completion */

    CMMD_sync_with_nodes();       /* ready to start */
    while(CMMD_get_global_or()) { /* loop until all done  */

      if (CMMD_msg_pending(CMMD_ANY_NODE, tag)) { /* poll for incoming msgs */
    CMMD_receive(0, tag, &bufferin, msg_len);
    recv_count++;
      }
    }

    /* print results */

    printf("PN %4d, msgs received = %6d \n", my_address, recv_count);
    total = CMMD_reduce_int(recv_count, CMMD_combiner_add);
  }
}
```

A.1.4 redist_nonblock

Global redistribution using asynchronous (nonblocking) functions

Each node generates $n = 100$ random integers in the range (0, **num_nodes**-1) and redistributes them in such way that a given node receives only the numbers which are equal to its node address. The global asynchronous “or” flag is used to test for completion.

```
#include <stdio.h>
#include <cm/cmmd.h>
#if CMMD_VERSION <= 20
#include <cm/cmmd-io.h>
#endif

main()
{ int msg_len=4, tag=0, rand_num, send_ok, n, recv_ok, bufferin;
  int send_count=0, recv_count=0, send_done=0, total;
  int my_address=CMMD_self_address(), num_nodes=CMMD_partition_size();

  /* set independent IO mode */

  CMMD_fset_io_mode(stdout, CMMD_independent);
  CMMD_fset_io_mode(stderr, CMMD_independent);
  if (my_address == 0) printf("\n Executing redist_nonblock ...\n");

  n = 100;                      /* each node will get 100 msgs. on average */

  srand(my_address);            /* initialize RNG */
  CMMD_set_global_or(1);          /* set global "or" to 1 (not completion) */
  while(!CMMD_get_global_or());   /* wait for setting to take place */

  /* send and receive loop */

  while(CMMD_get_global_or()) {   /* is everybody done ? */

    if (send_count < n) {         /* more sending ? */
      rand_num = (rand() >> 10) % num_nodes; /* generate random number */

      send_ok = CMMD_send_noblock(rand_num, tag, &rand_num, msg_len, NULL);
      if (send_ok == 0) send_count++;
    }
    send_done = (send_count == n);     /* sending done */

    if (CMMD_msg_pending(CMMD_ANY_NODE, tag)) {  /* poll for incoming msgs */

      recv_ok = CMMD_receive(CMMD_ANY_NODE, tag, &bufferin, msg_len);
      if (recv_ok == 0) {
    recv_count++;
    if (bufferin != my_address) {
      printf(" Error: received wrong msg: %d instead of %d\n",
          bufferin, my_address);
    }
      }
    }
    if (send_done && CMMD_all_msgs_done()) /* am I done ? */
      CMMD_set_global_or(0);
  }

  /* print results */

  printf("PN %4d, msgs received = %6d \n", my_address, recv_count);
  total = CMMD_reduce_int(recv_count, CMMD_combiner_add);
  if (my_address == 0) {
    printf("\n Total msgs generated (all nodes) = %8d\n", n*num_nodes);
    printf("\n Total msgs received (all nodes) = %8d\n", total);
  }
}
```

A.1.5 redist_nonblk_hndlr

Global redistribution using asynchronous (nonblocking) functions and a handler

Node 0 (master) generates n = (**num_nodes**-1)*100 random integers in the range (1, **num_nodes**-1) and redistributes them in such way that a given node receives only the numbers which are equal to its node address. A handler function counts the number of messages delivered. The global asynchronous "or" flag is used to test for completion.

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

#if CMMD_VERSION <= 20
#include <cm/cmmd-io.h>
#endif

#ifdef __STDC__
volatile
#endif
  int msgs_delivered = 0;

main()
{
  int tag=0, i, n, bufferin, recv_count=0, total;
  int my_address=CMMD_self_address(), num_nodes=CMMD_partition_size();
  CMMD_mcb mcb;
  int *rand_num;        /* Where to put the random nums.  */

  void my_handler();

  /* set independent IO mode */
  CMMD_fset_io_mode(stdout, CMMD_independent);
  CMMD_fset_io_mode(stderr, CMMD_independent);

  if (my_address == 0) {        /* Node 0: sender node */
    printf("\n Executing redist_nonblk_hndlr ...\n");
    CMMD_set_global_or(1);      /* set global "or" to 1 (not completion) */
    while(!CMMD_get_global_or()); /* wait for setting */
    CMMD_sync_with_nodes();     /* ready to start */
    srand(63924);               /* initialize RNG */
    n = (num_nodes-1)*100;      /* each node will get 100 msgs on average */

    rand_num = (int *) malloc(n * sizeof(int));

    for (i = 0; i < n; i++) {   /* send n random numbers to nodes */
     rand_num[i] = 1 + (rand() % (num_nodes-1)); /* generate random number */
     CMMD_send_async(rand_num[i], tag, &rand_num[i], sizeof(rand_num[i]),
             my_handler, 0);
    }

    /* the handler takes care of counting the number of msgs. being
       delivered */
    while(msgs_delivered != n);  /* wait until all msgs have been delivered */

    CMMD_set_global_or(0);      /* set global "or" to 0 (completion) */
    printf("\n Total msgs generated (node 0)= %8d\n", n);
    total = CMMD_reduce_int(recv_count, CMMD_combiner_add);
    printf("\n Total msgs received by nodes (1-- num_nodes-1)= %8d\n", total);
  }
  else {                        /* Nodes (1 -- num_nodes-1): recipients */
    CMMD_set_global_or(0);      /* node 0 will control completion */
    CMMD_sync_with_nodes();     /* ready to start */
    while(CMMD_get_global_or()) { /* loop until all done  */
      if (CMMD_msg_pending(CMMD_ANY_NODE, tag)) { /* poll for incoming msgs */
        CMMD_receive_block(0, tag, &bufferin, sizeof(bufferin));
        recv_count++;
      }
    }
    /* print results */

    printf("PN %4d, msgs received = %6d \n", my_address, recv_count);
    total = CMMD_reduce_int(recv_count, CMMD_combiner_add);
  }
}

void my_handler(mcb, arg)
CMMD_mcb *mcb;
int arg;
{
  /* free the mcb and add one to the counter of msgs. delivered */

  CMMD_free_mcb(*mcb);
  msgs_delivered++;
}
```

A.1.6 sort

Enumerate-pack sort (parallel radix)

Here's an example of a CMMD program written in the host/node programming style. This sort example performs the following operations for each bit of the key: All the elements with a 0 in that bit position are enumerated with a scan with add. Then a second scan does the same for elements with a 1. A scan with max (downward) broadcasts the sum of elements with a 0 to all nodes. This sum is added to the result of the second scan and these values are the new node addresses for elements with a 1 in the bit position. The new node addresses for elements with a 0 is just the enumeration obtained from the first scan. The initial number in each node is randomly generated.

Host Program:

```
#include <stdio.h>
#include <sys/types.h>
#include <cm/cmmd.h>

#if CMMD_VERSION <= 20
#include <cm/cmmd-io.h>
#endif#

main ()
{ int *numbers, *sorted, i, num_nodes, pid = getpid();

  printf("\n Executing host program... (pid=%d)", pid);

  CMMD_enable();
  num_nodes = CMMD_partition_size();

  numbers = (int *) malloc (num_nodes*sizeof(int));
  sorted = (int *) malloc (num_nodes*sizeof(int));

  CMMD_gather_from_nodes(numbers, sizeof(int));

  CMMD_gather_from_nodes(sorted, sizeof(int));

  printf("\n\n %8s   %11s\n", "numbers:", "after sort:");

  for (i = 0; i < num_nodes; i++)
      printf("\n %8d   %11d", numbers[i], sorted[i]);

  printf("\n All done \n");

}
```

Node Program

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

#if CMMD_VERSION <= 20
#include <cm/cmmd-io.h>
#endif

main()
{ int seed = 1234, msg_len=4, tag=0, tempo, number, k=0;
  int my_address=CMMD_self_address(), num_nodes=CMMD_partition_size();
  int mask_zero, mask_one, enum_one, enum_zero, sum, new_pos;

  CMMD_enable_host();

  srand(my_address*seed);                /* initialize RNG */
  number = (rand() >> 10) % num_nodes;   /* generate random number */

  CMMD_concat_elements_to_host(&number, sizeof(int));
```

```
for (k = 0; 1 << k < num_nodes; k++) {
  mask_one = number >> k & 1;
  mask_zero = ~(number >> k) & 1;
  enum_one = CMMD_scan_int(mask_one, CMMD_combiner_add, CMMD_upward,
                 CMMD_none, NULL, CMMD_inclusive) - 1;

  enum_zero = CMMD_scan_int(mask_zero, CMMD_combiner_add, CMMD_upward,
                 CMMD_none, NULL, CMMD_inclusive) - 1;

  sum = CMMD_scan_int(enum_zero + 1, CMMD_combiner_max, CMMD_downward,
                 CMMD_none, NULL, CMMD_inclusive);

  enum_one += sum;
  new_pos = enum_one*mask_one + enum_zero*mask_zero;

  CMMD_send_and_receive(CMMD_ANY_NODE, tag, &tempo, msg_len,
              new_pos, tag, &number, msg_len);
  number = tempo;
}

CMMD_concat_elements_to_host(&number, sizeof(int));
```

A.2 Fortran 77 Examples

Global redistribution using synchronous (blocking) functions

Node 0 generates *n* random integers in the range (0, **num_modes**-1) and redistributes them in such way that a given node receives only the numbers which are equal to its node address. A global asynchronous "or" flag is used to test for completion.

```
      program redist_block

      include "/usr/include/cm/cmmd_fort.h"

      integer n, msg_len, tag, i, rand_num, bufferin, seed, total
      parameter(tag = 0, msg_len = 4, n = 1000)
      integer my_address, num_nodes, msg_ok, ret_val, recv_count

      total =  cmmd_set_io_mode(6, cmmd_independent)
      my_address = cmmd_self_address()
      num_nodes = cmmd_partition_size()
      recv_count = 0

c      node 0: sender node

      if (my_address.eq.0) then

c     set global "or" to 1 (not completion)

         ret_val = cmmd_set_global_or(1)

c        wait for setting
         do while (cmmd_get_global_or().eq.0)
         end do

c        ready to start
         call cmmd_sync_with_nodes()

         seed = my_address

c        distribute (send) random numbers to nodes

         do i = 1, n
c           generate random number
            rand_num = 1 + int((num_nodes-1)*rand(seed))
            msg_ok = cmmd_send(rand_num, tag, rand_num, msg_len)
         end do
```

```
c        set global "or" to 0 (completion)
        ret_val = cmmd_set_global_or(0)
        print *," Total msgs generated (node 0)= ", n
        total = cmmd_reduce_int(0, CMMD_combiner_add)
        print *,"Total msgs received by nodes (1--num_nodes-1)=",total
        call flush(6)

      else

c      nodes (1 -- num_nodes-1): recipients

c     node 0 will control completion

        ret_val = cmmd_set_global_or(0)

c     ready to start
        call cmmd_sync_with_nodes()

c       loop until all done
        do while (cmmd_get_global_or().ne.0)
c        poll for incoming msgs
          if (cmmd_msg_pending(CMMD_ANY_NODE, tag).ne.0) then
             mesg_ok = cmmd_receive(0, tag, bufferin, msg_len)
             recv_count = recv_count + 1
          end if
        end do
        print *, "PN",  my_address, "  msgs received = ", recv_count
        total = cmmd_reduce_int(recv_count, CMMD_combiner_add)

      end if

      call flush(6)
      end
```

Simple I/O

Here's an example of a host/node program written in CM Fortran:

Host Program:

```
 program simple_io
      include "/usr/include/cm/cmmd_fort.h"

      integer nodes, i, data, msg_ok

      call cmmd_enable()
      nodes = cmmd_partition_size()
      print *, "start"
      call cmmd_bc_from_host(nodes, 4)

      call cmmd_service_loop()

      do 100 i = 0, nodes-1
      msg_ok =  cmmd_receive(i,0,data,4)
      print *, "PE ", i, " Sent ", data
 100  continue
      end
```

Node Program:

```
      program simple_io
      integer data, msg_ok, ret_code, dummy
      integer host

      include "/usr/include/cm/cmmd_fort.h"

      call cmmd_enable_host()

      call cmmd_receive_bc_from_host(dummy, 4)

      ret_code = cmmd_set_open_mode(CMMD_independent)
      write (*, *) "hello world"
      call cmmd_global_suspend_servers()

      data = cmmd_self_address() * 2
      host = cmmd_host_node()
      msg_ok =  cmmd_send(host,0,data,4)
      end
```

Appendix B

Non-RTS Parallel Memory Allocation

For some special-purpose applications, it is necessary to allocate parallel CM memory on a CM-5 with VUs without using the standard memory allocation and deallocation routines provided by the CM Run-Time System (CMRTS). An example of this is CMMD applications that include DPEAC subroutines.

Methods for allocating parallel memory without use of the CMRTS are described in the sections below.

B.1 The Parallel Stack and Parallel Heap

Parallel VU memory can be mapped into two general regions of memory, the parallel stack and the parallel heap.

Both the stack and the heap regions grow upward, toward higher memory addresses. Initially, no pages are allocated. When you allocate new space in these regions, it is as a stripe of memory across the physical memory of the four VUs. Thus, allocating a page of stack or heap actually allocates 4 pages of physical memory.

B.2 VU Regions

The stack and heap are each actually mapped as seven regions in memory in order to make the memory accessible in parallel or by individual VUs for the purpose of data or instruction access. (See the *DPEAC Reference Manual* for a description of VU space virtual addressing.)

By convention, pages of parallel memory are referenced by their *All-DP, Instruction Space* address. This is the address in the region of VU memory that causes all four VUs to execute a DPEAC instruction simultaneously. CMMD routines that take parallel memory addresses take such addresses (just as RTS routines do).

Whether coding in assembly (DPEAC) or C, you need to include this header file:

```
#include <cmsys/dp.h>
```

This header file defines a number of constants that are essential in navigating the VU memory regions. For example, the base of the parallel stack (in All-DP Instruction Space) is given by the symbol **DPV_STACK_INST_PORT_ALL** (**0x50000000**), and the base of the parallel heap region is given by the symbol **DPV_HEAP_INST_PORT_ALL** (**0x70000000**).

You can construct an address within these regions by adding a byte offset to these base addresses.

Important: Before you can access a stack or heap word, the memory region must have been expanded to include the address (that is, you must first allocate the memory before you can legally access it).

B.3 Expanding the Stack or Heap

When you want to expand the stack or heap, you make a CMOST system call to manipulate the pointer of the appropriate memory region. You can do this either from the partition manager or from a processing node. If you do this from a node, only one processing node must (and should) make the allocation call.

To access the appropriate CMOST routines, include the header file:

```
#include <cmsys/cm_memory.h>
```

The memory pointer system calls from the partition manager are:

```
CM_memaddr_t
  CM_set_dp_stack_ptr (CM_memaddr_t new_limit)

CM_memaddr_t
  CM_set_dp_heap_ptr (CM_memaddr_t new_limit)

CM_memaddr_t  CM_get_dp_stack_ptr ()

CM_memaddr_t  CM_get_dp_heap_ptr ()
```

The equivalent calls from the node are:

```
CM_memaddr_t
  CMPE_set_dp_stack_ptr (CM_memaddr_t new_limit)

CM_memaddr_t
  CMPE_set_dp_heap_ptr (CM_memaddr_t new_limit)

CM_memaddr_t CMPE_get_dp_stack_ptr ()

CM_memaddr_t CMPE_get_dp_heap_ptr ()
```

All of these routines return a **CM_memaddr_t** value, which is an *All-VU, Instruction Space* address, representing the current position of the memory pointer (in the case of the set routines, this is the value of the pointer after you have modified it). The value of the pointer is always one more than the highest allocated address in the memory region.

You cannot access allocated memory using the **CM_memaddr_t** values returned from these system calls, because they are in All-VU instruction space. You must translate this value into a *Single-VU Data Space* pointer, as described in Section B.4 below.

To use the set system calls, you pass in the highest address that you want to have allocated. The pointer value the call returns will always be greater than this value (unless there is insufficient memory remaining, in which case zero is returned), but it may not be exactly one more than the address you passed in.

Important: Don't make a "copy" of the stack or heap pointer and expect the copy to remain valid. Stack and heap memory can be allocated for other reasons than explicit system calls from your program. Thus, the stack and heap pointers can change without warning. You should always use the current value returned by the system calls mentioned above when determining the current size of the stack or heap.

If you want to deallocate parallel memory (in other words, shrink the stack or heap regions), call the appropriate **`set`** function with the new lower limit.

Note: CMOST currently does not allow the regions to shrink, and thus the above call will have no effect, and the current limit will be returned. Nevertheless, it is sensible to include deallocation calls, for compatibility with later software versions.

B.4 Translating Stack and Heap Addresses

You can change **`CM_memaddr_t`** values into valid data space addresses using the following C macro, which is defined in **`cmsys/dp.h`**:

```
data_address = TOGGLE_DPV_SPACE(instruction_address);
```

Note that the returned data space address is still an *All-VU* address. It cannot be used to read from memory, and if used to store to memory, the stored value will be written to all four VUs (broadcast).

You can change the data space address to point to a single VU by using one of the following macros:

```
VU_0_address = CHANGE_DP(data_address, DP_0);
VU_1_address = CHANGE_DP(data_address, DP_1);
VU_2_address = CHANGE_DP(data_address, DP_2);
VU_3_address = CHANGE_DP(data_address, DP_3);
```

The resulting addresses are pointers to single word/doubleword in stack or heap memory and can be used, for example, as a C pointer value to read or write memory values.

Note: Parallel memory, accessed by the node processor is always mapped with caching *disabled.* Thus, access to words/doublewords in the above fashion will be 2 to 3 times slower than normal cached accesses.

Also, all attempts to read/write parallel memory using pointers that are not word aligned will result in memory faults.

B.5 Using pmalloc to Allocate Memory

There is a sample C routine, **`pmalloc`**. An example of its definition and use can be found in the file:

```
/usr/examples/cmmd/hostless/c/mp_parallel_perf/parray.c
```

Index