The Connection Machine System

CM-5 User's Guide

Version 7.2, August 1993

Thinking Machines Corporation

First printing, August 1993

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

Contents

About This Manual

Objectives of This Manual

This manual is an introduction to using the Connection Machine CM-5 supercomputer. It describes the hardware and software that make up the system, and gives an overview of the commands and tools available to help you develop and run CM-5 programs.

Intended Audience

This manual is intended for people who want to write programs for and run programs on the CM-5 system. We assume that you have some familiarity with the UNIX operating system.

Revision Information

This is a new manual.

Organization of This Manual

Chapter 1 is an introduction to the CM-5.

Chapter 2 describes the basics of using the CM-5.

Chapter 3 describes how to use Prism, the CM-5's programming environment.

Chapter 4 is an overview of the CM-5's languages, libraries, and related software.

Appendix A covers some of the issues involved in porting programs from a CM-2 or CM-200 system to the CM-5.

Appendix B provides the source code for the sample program `primes.fcm`.

Appendix C is a glossary of UNIX and CM-5 terms used in the manual.

Related Documents

For in-depth information about the design of the CM-5, consult the *CM-5 Technical Summary*. Consult the other volumes of the CM-5 documentation set to learn more about many of the topics discussed in this manual.

See your Sun documentation for information about the UNIX operating system.

Notation Conventions

The table below displays the notation conventions used in this manual:

Convention	Meaning
`boldface`	UNIX and CMOST commands, command options, and file names.
`Ctrl-d`	Combinations of keystrokes are shown with a connecting hyphen. To type the `Ctrl-d` combination, for example, press the *d* key while holding down the *Control* key.
italics	Parameter names and placeholders in function and command formats.
`typewriter`	Code examples and code fragments.
`% bold typewriter` `typewriter`	In interactive examples, user input is shown in **`bold typewriter`** font 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

The Connection Machine CM-5 supercomputer combines high performance and ease of use for programmers working on large, complex, data-intensive applications.

This chapter begins by describing parallel programming on the CM-5, and then presents an overview of the CM-5's hardware, software, and I/O. For a more in-depth discussion of these topics, see the *CM-5 Technical Summary.*

1.1 Parallel Programming on the CM-5

The Connection Machine CM-5 supercomputer offers users high performance through a complete range of parallel processing approaches. The CM-5 supports both data parallel programming and message-passing programming:

- The CM's data parallel compilers (CM Fortran, C*, *Lisp) present the user with a global address space and a single thread of control.
- The CMMD communications library, callable from Fortran 77, C, and C++ (as well as from CM Fortran and C*) provides fast communication between independent tasks (or threads of control).
- Both models utilize the rapid synchronization and low-latency communication capability of the CM-5.

In the past, programmers of supercomputers were forced to choose between these two models. The CM-5, however, supports both; in fact, data parallel programs

and message-passing programs can run simultaneously, under timesharing, on a single partition of a CM-5.

1.2 The SPMD Programming Model

It is sometimes said that virtually all successful parallel programs in existence today are written in the SPMD (Single Program, Multiple Data) style of programming: a style in which a single program runs on a multi-processor system, with each processor acting on its share of the program's multiple data items.

Both data parallel programming and message-passing programming on the CM-5 use the SPMD model. The differences are:

- whether your program takes primarily a global view or a local view of the system
- the mechanisms by which the actions of the CM-5's processing nodes are coordinated

1.2.1 Data Parallel Programming

Data parallel programs take a global view of the system. A single program executes on the control processor. This program controls all the processing nodes, requesting synchronized computation, communication, and I/O as needed. These programs are written in the CM's data parallel languages, such as CM Fortran or C*, using the data parallel compilers and run-time system to control data layout and inter-processor communication and synchronization. They also make use of the CM's specialized parallel library routines and I/O functionality.

With data parallel programming, the programming nodes work in synchrony. For example, let's consider a finite difference code that needs to perform one operation on its boundary elements and another on interior elements. Code written to deal with this case in a data parallel language would look something like:

```
where (boundary_elements)
   do_a
elsewhere
   do_b
end where
```

On a CM-5, both branches can execute simultaneously. That is, each processor decides, for each data element, whether to fetch and execute the instructions for the `where` branch or for the `elsewhere` branch. All processors synchronize at the beginning and end of the `where` block; but within the block, each computes asynchronously with respect to the others. Most important, the synchronization is still automatic, under control of the system software, and presents no problem to the programmer.

This single flow of control, so similar to that of a standard "serial" program, makes data parallel programs the easiest of parallel programs to debug, and helps account for the popularity of this programming style.

For more information on data parallel programming, we recommend that you consult the documentation for CM Fortran or C*.

1.2.2 Message-Passing Programming

Message-passing programs take a node-level view of the system. Again, a single program executes; but in this programming style, a separate copy of the program executes independently on each node. The nodes divide tasks and data among themselves according to the needs of the application; they may stay closely synchronized or become completely asynchronous. All communications and synchronization, as well as data layout, are under the application's explicit control.

The message-passing programming style is most useful when an application requires the dynamic allocation of tasks or data. Such applications typically use a class of algorithms known as node-expansion algorithms: examples are divide and conquer algorithms, branch and bound algorithms, asymmetric traveling salesman problems, and tree search problems.

On the CM-5, message-passing programs use the CM's communications library, CMMD. CMMD functions can be called from C, C++, Fortran 77, CM Fortran, or C*; the ability to use standard high-level languages is appreciated by users who have existing programs that they wish to port to a parallel supercomputer.

1.3 Scalable Computing

The ability to choose among programming models is an important feature of the CM-5, since it lets users choose the technique that is best, not only for their application, but for each part of their application. Also important is the Connection Machine's support for Scalable Computing, and its provision of tools geared specifically to the needs of its users, such as the Prism programming environment and the CMAX Fortran 77-to-CM Fortran translator.

Connection Machine data parallel programming has always been inherently scalable. Because the CM's data parallel software lays out data arrays at run time, a single program can run on any size Connection Machine, with computation and communication patterns optimized for machine size. Now, the CM-5 allows message passing to be scalable as well.

1.4 CM-5 Hardware

The architecture of the CM-5 is optimized for data parallel processing of large, complex problems. Figure 1 shows this architecture.

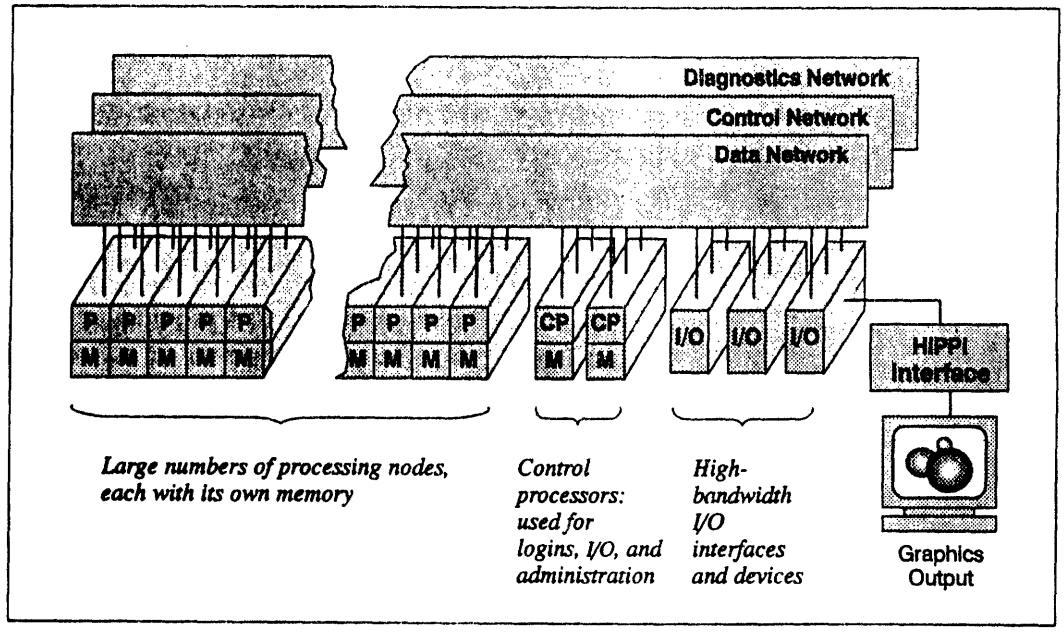

Figure 1. Organization of the CM-5.

1.4.1 Processors

A CM-5 has many parallel processing nodes, each with its own memory. Nodes can fetch from the same address in their memories to execute the same instruction, or from individually chosen addresses to execute independent instructions.

These processing nodes are supervised by a *control processor*. The system administrator can divide the nodes into groups, known as *partitions*. There is a separate control processor, known as a *partition manager*, for each partition. Each user process executes on a single partition, but can exchange data with processes on other partitions.

Control processors that don't manage partitions manage the system's I/O devices and interfaces. This organization allows a process on any partition to gain access to any I/O device, and ensures that access to one device does not impede access to other devices.

1.4.2 Vector Units

The CM-5 can optionally contain high-performance arithmetic hardware, known as *vector units*. These vector units use wide data paths, deep pipelines, and large register files to improve peak computational performance. Users who have CM-5s with vector units typically compile their programs for, and run their programs on, the vector units.

1.4.3 Networks

Every control processor and parallel processing node in the CM-5 is connected to two communication networks, the *Data Network* and the *Control Network*.

In general, the Control Network is used for operations that involve all the nodes at once, such as synchronization operations and broadcasting. The Data Network is used for bulk data transfers where each item has a single source and destination.

A third network, the Diagnostics Network, is visible only to the system administrator; it keeps tabs on the physical well-being of the system.

External networks, such as Ethernet and FDDI, can also be connected to a CM-5 via the control processors.

1.5 CM-5 Software

As suggested above, the Connection Machine CM-5 provides software to support both data parallel and message-passing programs. Figure 2 diagrams the current CM-5 software offerings.

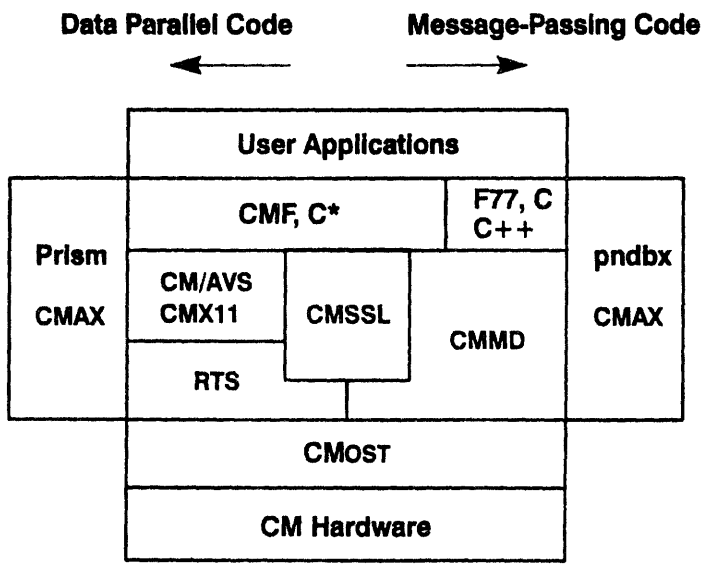

Figure 2. CM-5 software.

1.5.1 Data Parallel Software

- The CM-5 currently offers three high-level languages for data parallel programming: CM Fortran, C*, and *Lisp. These languages are nearly identical to their CM-200 counterparts. A few implementation-specific differences exist: these are detailed in Section 4.1 for CM Fortran and Section 4.2 for C*. Readers interested in *Lisp should consult the document *Porting to CM-5 *Lisp.*

- Data parallel libraries include CMSSL (the CM Scientific Software Library) and CMX11 (a visualization library that provides parallel extensions to the X11 standard). Section 4.3 briefly introduces CMSSL; Section 4.4.4 introduces CMX11.

- CM/AVS is a Graphical User Interface for visualization that links the computing power of the CM-5 with the convenience of a specialized graphics workstation. It is described in Section 4.4.3.

1.5.2 Message-Passing Software

Message-passing programs on the CM-5 can be written in C, C++, Fortran 77, CM Fortran, or C*. They are supported by a message-passing library, CMMD. This library and its associated support utilities in the operating system are summarized in Section 4.5.

1.5.3 Assemblers

The assembly language for the CM-5 is DPEAC; its assembler is **dpas**. The CM Fortran and C* compilers compile programs into DPEAC code.

The DPEAC language is a superset of the SPARC **as** assembly language. Sun's C, C++, and Fortran 77 compilers compile programs into **as**.

On a CM-5, **as** statements can control the SPARC microprocessors in the nodes and in the partition manager; DPEAC statements can control both the SPARC and the vector units.

The **dpas** assembler reads DPEAC sources and outputs UNIX object (**.o**) files. It does this by preprocessing its source, translating the DPEAC-specific statements into SPARC assembly source, and running the resulting text through the **as** assembler. Any SPARC assembly statements are passed unchanged to the **as** assembler.

Because the **dpas** assembler contains an excellent preprocessor for both **as** and DPEAC statements, it is sometimes very useful as a preprocessor for an **as** program.

For full information on DPEAC and **dpas**, see the *DPEAC Reference Manual.*

1.5.4 CMAX

CMAX — the "CM Automated X-lator" — assists the conversion of standard Fortran 77 into CM Fortran. CMAX provides a convenient migration path for serial programs onto the CM-5. Software may be maintained in either Fortran 77 or CM Fortran. CMAX is described in Section 4.6.

1.5.5 The CMOST Timeshared Operating System

The CM-5 operating system, CMOST, is an enhanced version of the UNIX operating system. As such, it enables the CM-5 to interact efficiently with other devices in a heterogeneous, networked computing environment, while at the same time managing the interactions of CM-5 system components and managing the timeshared execution of multiple parallel programs.

As a resource on a heterogeneous computing network, CMOST provides:

- standard UNIX utilities, user interfaces, protection, and security
- support for all standard UNIX-based communications protocols
- the ability to access files and exchange data with other systems

As a resource within the CM-5 system, CMOST provides:

- fast interprocessor communications
- the parallel operations required for best utilization of processing hardware, especially the array of processing nodes
- central administration and resource management to support all CM-5 computation and I/O facilities
- timeshared execution of parallel programs

1.5.6 Prism Programming Environment

Prism is a programming environment that integrates debugging, profiling, and other useful tools in a convenient windowed environment. A graphical interface is available from terminals and workstations that are running the X Window System. A command interface is available from other terminals. Although designed

primarily for use with data parallel programs, Prism can also be useful for other program development. Prism is described in Chapter 3.

1.6 I/O on the CM-5

1.6.1 Hardware

The CM-5 supports a wide range of I/O devices, including:

- *Scalable Disk Array (SDA).* The SDA is an extremely high-performance, highly expandable disk storage system packaged within the CM-5 cabinetry.
- *CM HIPPI interfaces.* CM-HIPPI and CM5-HIPPI are bus interface controllers that are designed to transfer data at a high speed according to the ANSI HIPPI draft standard. The interfaces are primarily intended to link the CM-5 and its storage devices to other supercomputer systems.
- *CM-2/200 I/O devices.* For application portability, the family of CM-2 and CM-200 peripherals continues to be supported on the CM-5. These devices reside on Thinking Machines' proprietary CMIO bus, which is connected to the Data Network. These peripherals are:
  - *DataVault*, a high-performance, disk-based mass storage system
  - *CM-IOPG*, an I/O controller that provides four ports for connection to SCSI-based devices, such as cartridge tape drives

1.6.2 Software

CM-5 software supports three file systems:

- SFS, the Scalable File System, provides access to the CM-5's high-performance Scalable Disk Array (SDA).
- CMFS, the Connection Machine File System, provides access to all I/O devices that are shared by the CM-5, CM-2, and CM-200. These devices (sometimes called the CMIO-bus devices) include the DataVault mass storage system, the CM-HIPPI, and VME-based devices.

- The standard NFS-mounted UNIX file system.

The CM Fortran Utility Library, C* I/O procedures, and CMMD I/O routines allow applications to access disk files in any of these file systems. These procedures allow applications to open, close, truncate, and seek within files. They also allow applications to read and write data in parallel streams between the processing nodes and the SDA or DataVault.

In addition, UNIX files can often be accessed by standard UNIX commands, while special CMFS library procedures enable you to manipulate files stored on devices accessible via VME or CM-HIPPI.

This flexibility in file access, together with the fact that the CM-5 writes files in canonical UNIX order, allows the CM-5 to share data with other machines in a heterogeneous environment.

Chapter 2

The Basics

This chapter covers the basic procedures you need to know in order to create, compile, execute, and debug programs on the CM-5. If you have used the UNIX operating system, much of what is covered in this chapter will be familiar to you. If you are not familiar with UNIX, you should begin by reading an introductory book about the operating system.

All the procedures discussed in this chapter can be carried out from within Prism, the CM-5's programming environment; Chapter 3 describes Prism.

Your CM-5 should have sample programs you can use, if you want to try running a program on the computer before writing one yourself. Typically, these programs should be in the directory `/usr/examples`; if this directory does not exist, check with your system administrator for the location of sample programs on your system.

In this guide, we refer to a program called `primes`, which is a simple data parallel CM Fortran program for calculating and displaying prime numbers. By default, this program, along with its source code, is in the directory `/usr/examples/prism` on a partition manager. Check with your system administrator if it isn't there. We provide the source code in Appendix B, in case you can't find it on-line.

We suggest that you copy the source file, `primes.fcm`, to your home directory, from which you can compile it and use it within Prism, following the directions we provide in this chapter.

2.1 The User's View of the CM-5

As we mentioned in Chapter 1, a CM-5 has one or more partition managers, each of which controls some number of parallel processing nodes. These partitions operate independently of each other, although data can be passed back and forth between partitions. Your system administrator determines the exact configuration of your site's CM-5.

Each partition manager is a host on a network. You can log in to a partition manager over the network to gain access to the nodes that the partition manager controls, as well as to all the CM-5's I/O devices. Your system administrator can determine which users can log in to a given partition manager.

In Figure 3, the CM-5 has two partition managers, named Mars and Venus; each is currently managing a partition of 256 nodes; this CM-5 has the optional vector-unit hardware, so each partition can also be viewed as containing 1024 vector units. The system also has control processors managing some I/O peripherals, and another control processor that is dedicated as a system console.

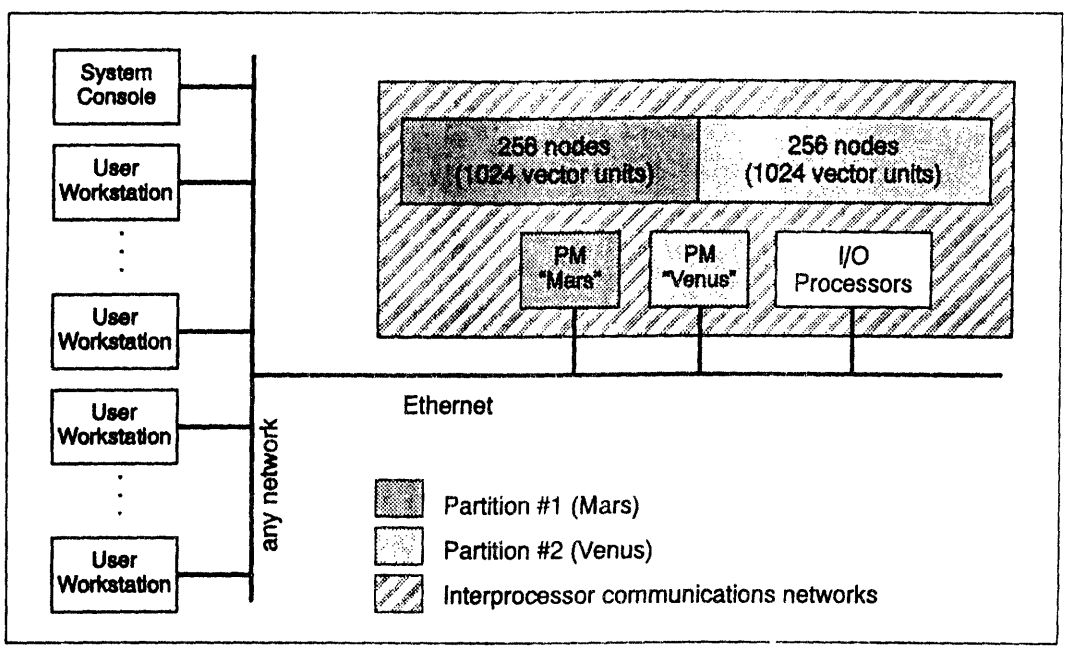

Figure 3. A sample CM-5 system.

You execute your program from the partition manager. The partition manager downloads code to the processors and broadcasts identical memory maps to each processor (node or vector unit). The processors then execute the code, each acting on its own data and executing computations and branches accordingly.

Multiple users can be logged in to a partition manager at the same time, running processes under timesharing. You can also issue commands over the network to submit your program to a batch queue for execution.

2.2 Gaining Access

To gain access to a partition manager, you must know its name. Find out the names of partition managers from your system administrator. The system administrator can also tell you if you have permission to use the partition manager.

From your terminal or workstation on the network, you can gain access to the partition manager via the UNIX command **rlogin** or **rsh**.

2.2.1 The rlogin Command

Use the **rlogin** command to log in to a partition manager remotely across the network. To log in to the partition manager Mars, for example, issue the command:

```
% rlogin mars
```

You will then see messages from the partition manager; it will also ask you to confirm your terminal type. For example,

```
term(xterm)
```

asks if you are using an X terminal. If the terminal type is correct, simply press your **Return** key. If it isn't, enter the correct terminal type and press **Return**. (If you don't know your terminal type, ask your system administrator for help.)

You should then see the UNIX/CMOST prompt:

```
%
```

You are now logged in to the partition manager. You can issue any CMOST command and execute programs. For example,

```
% primes
```

executes the program **primes**.

To log out of the partition manager, issue the command

```
% logout
```

2.2.2 The rsh Command

Use the `rsh` command to execute a program or issue a CMOST command without logging in to the partition manager. The `rsh` command creates a "remote shell" on the partition manager. Simply specify the name of the partition manager, followed by the command or program name, on the `rsh` command line. For example,

```
% rsh mars primes
```

executes the program `primes` on Mars.

After the program executes, you are returned to your local UNIX shell.

2.3 Writing and Compiling Your Program

You can create your program as you normally do, on your local computer.

If you are creating a C* or CM Fortran program, you can compile it on the partition manager, or on any computer that has CMOST and the C* or CM Fortran compiler installed. To compile the `primes` program from the source file `primes.fcm`, for example, issue this command:

```
% cmf -cm5 -o primes primes.fcm
```

If you are creating a C or Fortran 77 program, you can also compile it on the partition manager, or on any computer that has a `cc` or `f77` compiler.

Generally it's a good idea not to compile on the partition manager (if you have a choice), to avoid overloading it.

2.3.1 Linking

A CM-5 executable program actually consists of two programs, one that executes on the partition manager and the other that executes on the CM processors. CMOST uses a special linker, `cmld`, to create this executable program. If you are developing your program in a high-level language, you typically don't need to deal with `cmld`. You would use it if, for example, you were creating a C* program that calls CM Fortran subroutines.

2.4 Executing Your Program

As shown above, you execute your program as you normally would on a UNIX system: you specify the name of the executable program at the UNIX prompt, or as an argument to the `rsh` command. The program then runs under timesharing on the CM-5.

2.4.1 Providing Accounting Information

On some CM-5 systems, users are required to provide an account ID when running a program, so that time on the system can be charged to the correct project. Obtain the account ID from your system administrator. You then specify this account ID before running a program by setting the UNIX environment variable `CM_ACCOUNT_ID` to your number. For example:

```
% setenv CM_ACCOUNT_ID 1234
```

To avoid having to reissue this command constantly, you can put it in your `.cshrc` file, so that it is executed automatically when you log in or issue an `rsh` command.

Note that this mechanism doesn't work if you are executing in batch mode; see below.

2.4.2 Executing in Batch Mode

You can also run your program in batch mode. You can use either the standard UNIX batch command, the NQS batch system, or (if it is available at your site) the Distributed Job Manager (DJM).

The UNIX Batch Commands

The CM-5 supports the standard UNIX batch commands **at** and **batch**. For example,

```
% at 0815am Jan 24
at> primes
at> Ctrl-d
```

This causes the program **primes** to be executed at 8:15 a.m. on January 24th. *Ctrl-d* means "press the *d* key while holding down the *Control* key." This is the way to signal the end of information you are providing to the standard input.

The NQS Batch System

NQS is a batch system that your system administrator can set up to control the execution of programs on the CM-5. In it, you submit a request to a queue, which either runs on a partition manager or feeds requests to a queue on a partition manager. A request consists of one or more programs. When your request reaches the head of the queue, it is executed. Queues can be configured in various ways: for example, so that they are active only at certain times, or so that they accept requests only from certain users. A partition manager can be set up so that it accepts requests only from NQS queues, or so that requests from the queue timeshare with other processes running on the partition.

Use the **qsub** command to submit a request to a queue. Specify the name of the queue on the command line, using the **-q** option. You can specify the name of the program from the standard input, as with the **at** command. For example, if the queue's name is **b_queue**, you could type this:

```
% qsub -q b_queue
  primes
  Ctrl-d
```

Alternatively, you can put the program's name in a script file. This is especially useful if you have multiple programs to run, or you have CMOST commands you

want to issue along with the program. If the script file's name is **/myname/primes_script**, you would issue the command as follows:

```
% qsub -q b_queue /myname/primes_script
```

Use the **-I** option to specify your account ID, if one is required; obtain the account ID from your system administrator.

To find out how to obtain status information from NQS, see Section 2.7.3. For a complete discussion of NQS, see the manual *NQS for the CM-5*.

DJM (Distributed Job Manager)

DJM (Distributed Job Manager) is a batch queuing system and interactive job manager available on some CM-5s. It is similar to NQS: you submit a request to a queue for execution, and queues can be configured in various ways. Check with your system administrator to find out if DJM is available at your site.

NOTE: If DJM is running at your site, you should typically use it to run your CM-5 programs. Otherwise, you might have difficulty obtaining the resources to run your program.

Use the **jsub** or **jrun** command to submit a job. You can specify the name of the program from the standard input, or in a script file. For example, if the script file's name is **/myname/primes_script**, you would issue the command as follows:

```
% jsub /myname/primes_script
```

If you don't include the appropriate options (see below), DJM then responds by asking questions about the job:

```
Number of processors (32)?
Estimated CPU time (5min)?
Estimated memory (128M)?
```

(The defaults in parentheses may differ at your site.)

Make your best estimates of the resources your job will need to run. If your estimates are too low, DJM will not effectively schedule your job; if they are too high, the job may be delayed unnecessarily, waiting for the resources to be available. The actual resources consumed by your job are printed at the end of the output file when the job finishes. You can use this information to provide better resource estimates the next time you run your job.

When you have answered DJM's questions, it prints a job ID:

```
Job submitted successfully. Job id is 43.
```

Output from the job is written to the file *myjob*.o*nnn*, where *myjob* is the name of the program that ran and *nnn* is its job ID. You can override this by specifying the **-stdout** option; this sends output to the standard output.

Use the **-queue** option to specify the queue to which the job is to be submitted. If you omit the option, DJM submits the job to the first queue that meets the requirements for your job. You can also use the **-nproc**, **-cputime**, and **-mem** options to specify your job's resources on the command line, rather than having to respond to DJM's questions.

To find out how to obtain status information from DJM, see Section 2.7.4. For complete information on DJM, see your local DJM documentation.

2.5 Debugging Your Program

The standard way of debugging a data parallel program on the CM-5 is to use Prism, as described in Chapter 3. Prism supports the CM Fortran, C*, C, and Fortran 77 languages.

Debugging of messsage-passing programs is different from conventional debugging, since you may need to look at the status of the program in a specific node, or group of nodes. To debug the node program, you can use the CM-5 node-level debugger **pndbx**, which you can run in conjunction with Prism or with a debugger like **dbx** or **gdb**.

2.5.1 The Errors File and Core Files

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

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

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

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

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

```
CMOST: yellow interrupt
```

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

If your program was using the CM-5's vector units, additional error files are created in the directory from which you ran the program. They are called `CMTSD_dp.pn`*X*`.`*pid*, where *X* is the node identifier and *pid* is the process ID. These files contains an ASCII dump of the contents of every register on the vector units, for every node that has a different error.

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

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

You can examine these node core files using **pndbx**, the CM-5's node-level debugger. For information on how to do this, see the *CMMD User's Guide*.

If you don't want node core files generated, set the environment variable `CM_NO_PN_CORE` (if you are running the C shell) to any value. For example:

```
% setenv CM_NO_PN_CORE 1
```

You can also issue this command:

```
% limit coredumpsize 0
```

Issuing this command prevents the creation of core files from any program, whether or not it runs on the CM-5.

To re-enable the generation of CM-5 core files, unset the environment variable:

```
% unsetenv CM_NO_PN_CORE
```

2.5.2 Fortran Tracebacks

When a Fortran or CM Fortran program dies, it may generate a traceback. The traceback file will be called *prog*`.trace`, where *prog* is the name of your program. If there are multiple tracebacks in the file (because the program crashed multiple times), the last traceback is the most recent; it tells you the routine that was executing on the partition manager when the program died.

Note, however, that the information may not be especially valuable, since the partition manager is not necessarily synchronized with the nodes. For example, the partition manager may continue working for a while before an error status from the nodes is propagated to it and the program halts. Therefore, the routine or instruction that is executing on the partition manager when the nodes die may have nothing to do with the error.

2.6 Timing a Program

CMOST provides a timing utility that lets you determine how much time any part of a program takes to execute on the nodes. The timer consists of a set of instructions that you insert at the appropriate places in your program.

Prism also provides performance data about your program. See Chapter 3.

The timing utility has these features:

- It can be used in C* or CM Fortran programs.

- A timer calculates total elapsed time used by the process (or any part of it) and the total amount of time the CM processors are active, with microsecond precision.
- Multiple timers can be active at the same time.
- Timers can be nested. This allows you, for example, to start one timer that will time the entire program, while using other timers to determine how different parts of the program contribute to the overall time.
- You can have up to 64 timers running in a program.

To get timings for the **primes** program, you would include these calls at the appropriate places in the code:

```
CALL CM_TIMER_CLEAR(0)              ! Initialize timer 0
CALL CM_TIMER_START(0)              ! Start timer 0
CALL CM_TIMER_STOP(0)               ! Stop the timer
CALL CM_TIMER_PRINT(0)              ! Print the results
```

For complete information on using the timing utility from CM Fortran or C*, see the user's guides for those languages.

CMMD provides comparable functions for use within message-passing programs. See the *CMMD User's Guide* for more information.

2.7 Getting Information

CMOST provides commands you can use to find out information about the CM-5 and processes running on it. It is especially important to be able to get information about the CM-5's configuration, since the system administrator can easily reconfigure it to add or remove partition managers, change the number of nodes in a partition, and so on.

2.7.1 Finding Out about the CM-5's Configuration

To find out the general configuration of a CM-5, log in to a partition manager and issue the command **cmpartition**:

```
% cmpartition
```

The output might look like this:

```
Name      Partition Manager       Size  State
mars      mars.think.com          128   ACTIVE
venus     venus.think.com         128   ACTIVE
```

This describes a CM-5 system with two partition managers, Mars and Venus, each of which controls 128 nodes.

For more complete information, use the `list -l` option:

```
% cmpartition list -l
```

```
CM System "G1"
     256 Processors
     2 Partition Managers
         mars.think.com
         venus.think.com
Available PN Ranges:  All PNs in use
IOP Addresses  480

Name      Partition Manager       Size  State       Nodes     Description
mars      mars.think.com          128   ACTIVE      0-127     Batch only
venus     venus.think.com         128   ACTIVE      128-255
```

This provides the additional useful information that Mars is set up to run batch requests only.

2.7.2 Finding Out about the Status of the System

To find out information about the partition and the programs running on it, use the CMOST command **cmps**, which is modeled after the UNIX command **ps**. The command provides information about the partition on which you execute it. The output looks like this:

```
32 PN System, 29168K mem. free, 6160K VU mem. free, 2 procs, TS-9/13/92-23:00
Daemon up: 1 day,  2:57

USER   PID  CMPID   TIME  TEXT   ILH    ILS   IGS    IGH   VUS   VUH    COMMAND
mob    *900    1  234:13  896K  254K    56K    0K     0K   64K    8K    primes
rhv    7900    0   16:36  200K  230K    56K    0K     0K   64K    0K    wil
```

The first line of the **cmps** output provides general information about the partition, including the number of processing nodes (referred to as *PNs*) it contains and the amount of memory available on the nodes and the vector units.

The columns give information about each process:

- The **PID** and **CMPID** fields give the process's SunOS and CMOST process-ID number.
- The **TIME** field indicates the amount of time that the CMOST timesharing daemon has made available to the process, regardless of whether the process actually used the nodes.
- The **TEXT** field indicates the total amount of program memory allocated to the process.
- The fields **ILH**, **ILS**, **IGS**, **IGH**, **VUS**, and **VUH** give specifics on stack and heap memory used by the process. Note these points:
  - The numbers reported are the maximum for any single node or (in the case of VUS and VUH), any single vector unit. Thus, in the case of VUS and VUH, the total amount of memory taken up per node is 4 times the total of these two fields.
  - Memory is allocated uniformly across the nodes, even in message-passing processes.
  - To calculate the total amount of memory used by a process, you must multiply by the number of nodes or vector units in the partition on which it is running.

  For more details on these fields, see the **cmps** man page on-line.
- The **COMMAND** field gives the name of the program being executed.

To find out information about the partition manager and the programs running on it, use the standard UNIX command **ps**.

If you are running under the X Window System, you can use the **xcmps** command to get a graphical display of CM-5 usage on your partition. Sample output is shown in Figure 4. By default, the display is updated every 10 seconds. For more details, see the **xcmps** man page on-line.

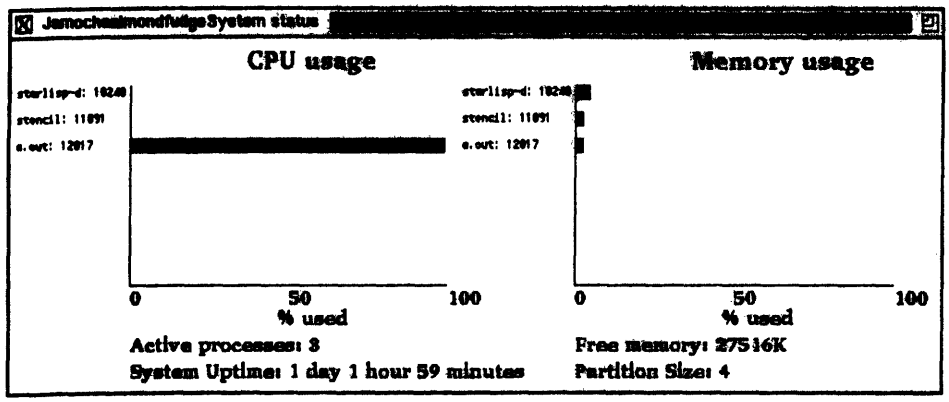

Figure 4. Output of the xcmps command.

2.7.3 Finding Out about the Status of NQS

To find out information about NQS, issue the command `qstat` on a partition manager. Use the -u option to obtain information about requests from a specific user. For example,

```
% qstat -u sharon
```

might produce this output:

```
------------------------------------------------------------
NQS Version: 2      REQUESTS on diamond
------------------------------------------------------------
REQUEST      NAME    OWNER    QUEUE    PRI    NICE    CPU      MEM      STATE
0●diamond. primes   sharon   pn_128    32      30    3600    UNLIMITED RUNNING
```

This shows that user Sharon has one request running on the queue `pn_128`. The other fields give information about characteristics of the request.

To find out information about batch queues, use the -b option. For example,

```
% qstat -b
```

might produce this response:

```
------------------------------------------------------
NQS Version: 2      REQUESTS on diamond.think.com
------------------------------------------------------
QUEUE NAME    STATUS    TOTAL   RUNNING  QUEUED   HELD  TRANSITION
-------------------------------------------------------------------
q1            AVAILBL    1       1/1      0        0       0
q2            AVAILBL    2       2/10     0        0       0
```

This indicates that there are two queues on the partition manager **diamond**; they are named **q1** and **q2**. The **TOTAL** field tells you the number of requests in the queue. The other fields give information about the status of the requests in the queue.

2.7.4 Finding Out about the Status of DJM

Use the **jstat** command to find out information about the Distributed Job Manager (DJM). With no arguments, **jstat** lists the current jobs. For example:

```
% jstat
USER       JID MACH  SERV  PROCS   TIME    MEM  STATUS    COMMAND
sharon      43 cm5   ea     32     0:02   512M  Running   myjob
mark        24 cm5          256   240:00   600M  Que npr   rdt
```

This indicates that user Sharon has a job running on 32 processors of partition manager **ea** of a CM-5; its job ID is 43. The **PROCS**, **TIME**, and **MEM** fields indicate the actual number of processors and amount of CPU time and total amount of memory being used.

User Mark's job is queued; in this case, the **PROCS**, **TIME**, and **MEM** fields indicate the number of processors and amount of CPU time and memory requested when the job was submitted. The status **npr** indicates that the job is not running because the requested number of processors is not available.

Another use of the `jstat` command is to find out the available queues and the resource limits associated with them. Use the `-limits` option to obtain this information. For example:

```
% jstat limits
QUEUE     SRCH  MACH  TIME    MEM    NPROC   DED  FOR  UQUE  URUN  TRUN  TMEM
foreign     50  cm5   20min   128M   unlim   no   yes   999   999   999  unlim
default    100  cm5    1hr     2G     512   no   no      4     4     8  unlim
big        100  cm5   unlim   unlim  unlim   no   no      2     2     3  unlim
ded        100  cm5   unlim   unlim  unlim   yes  no      1     1     2  unlim
```

In the output:

- **SRCH** indicates the order in which DJM searches the queues to determine in which one a job will run. It searches from the lowest-numbered queue to the highest, and submits the job to the first queue that meets the job's requirements.
- **TIME**, **MEM**, and **NPROC** give the maximum usage characteristics of the queue.
- **DED** specifies whether or not the queue is for dedicated jobs (that is, jobs that require exclusive use of the partition).
- **FOR** indicates whether the queue is for "foreign" jobs — that is, jobs that are run directly on the machine without going through DJM.
- **UQUE** and **URUN** indicate the maximum number of jobs an individual user can have queued and running in this queue.
- **TRUN** and **TMEM** indicate the total number of jobs that can be running concurrently in this queue, and the total amount of memory they can consume.

2.8 Obtaining On-Line Documentation

The CM-5 has a considerable amount of on-line documentation available to users. See Section 3.7.1 to find out how to view this documentation from within Prism.

From outside Prism, use the standard UNIX **man** command to view the manual page for a UNIX or CMOST command or routine.

Chapter 3

Using Prism

The Prism programming environment is the standard way in which users interact with the CM-5 in developing and executing their data parallel programs. Prism provides an integrated graphical environment in which you can:

- execute the program
- debug the program
- analyze the program's performance
- visualize data from the program
- obtain on-line documentation for the CM-5
- ... and much more

This chapter gives an overview of how to use Prism. For complete information on Prism, see the *Prism User's Guide* and *Prism Reference Manual.*

In this chapter we use the sample data parallel program `primes` to illustrate many of Prism's capabilities. If you have the program (and a CM-5) available, you can follow along on-line as we help you get started. To compile the program for debugging via Prism, specify the `-g` option; note that compiling with `-g` causes performance to be artificially slow. To compile the program to collect performance analysis data, specify the `-cmprofile` option. See Section 2.3.

3.1 Starting Prism

Prism runs under the X Window System.

To start Prism, first log in to a partition manager as described in Chapter 2. For example:

```
% rlogin mars
```

Start X on the partition manager as you normally do; if you're not familiar with X, ask your system administrator for help. Make sure your `DISPLAY` environment variable is set for the terminal or workstation from which you are running X. For example, if your workstation is named Valhalla, you can issue this command (if you are running the C shell):

```
% setenv DISPLAY valhalla:0
```

To start up Prism, issue the command `prism` at your UNIX prompt; you can include the name of your executable program as an argument. Thus,

```
% prism primes
```

loads the `primes` program. (This assumes that you are starting Prism from the directory in which `primes` is located.) The main Prism window then appears, as described below.

If you don't have an X terminal or workstation, you can run a non-graphical version of Prism; specify the `-c` option on the command line. This version of Prism is discussed in Section 3.10.

3.2 A Tour of Prism

Figure 5 shows the main window of Prism with the `primes` program loaded. In Prism, you can operate with a mouse, use keyboard equivalents of mouse actions, or issue keyboard commands.

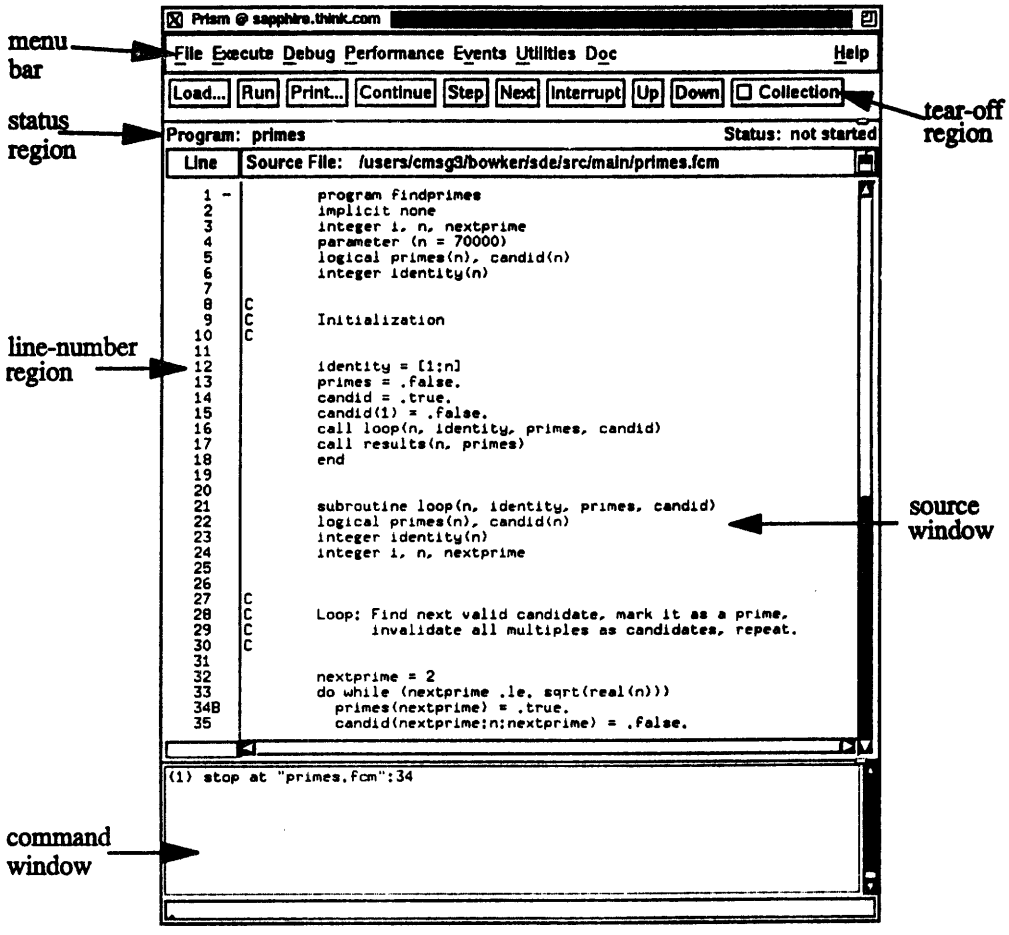

Figure 5. Prism's main window.

Left-clicking on items in the **menu bar** displays pulldown menus that provide access to most of Prism's functionality.

You can add frequently used menu items and commands to the **tear-off region**, below the menu bar, to make them more accessible; many items are there by default.

The **status region** displays the program's name and messages about the program's status.

The **source window** displays the source code for the executable program. You can scroll through this source code and display any of the source files used to compile the program. When a program stops execution, the source window updates to show the code currently being executed. You cannot edit the file in the

source window; instead, you call up a separate editor if you want to make changes to your code.

The **line-number region** is associated with the source window. You can click to the right of a line number in this region to set a breakpoint at that line. In Figure 5, a breakpoint is set at line 34. The *execution pointer* (>) in the line-number region shows the line at which the program is currently stopped. The *scope pointer* (-) shows the beginning of the scope that Prism uses in interpreting the names of variables. The symbol * appears when the execution pointer and the scope pointer are at the same line.

The **command window** at the bottom of the main Prism window has two areas. The *history region*, at the top, displays messages and output from Prism. You can type commands on the *command line* at the bottom of the window, rather than use the graphical interface.

3.3 Executing a Program

Once `primes` is loaded, you can execute it from within Prism. The simplest way to do this is to click on **Run** in the tear-off region of the main Prism window. (You can also choose the **Run** selection from the **Execute** menu, or issue the `run` command on the command line.)

After you have clicked on **Run**:

- The status changes to **running**.
- Many menu items are grayed out, indicating that they are temporarily unavailable.
- An Input/Output window appears; when the program has finished execution, the results will appear here. Figure 6 shows what it will look like for `primes`.

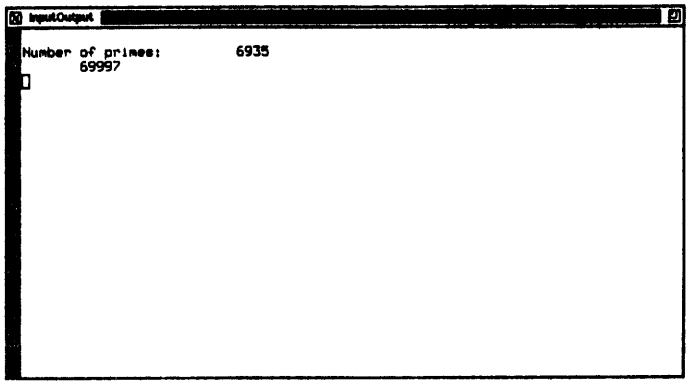

Figure 6. The Input/Output window.

3.3.1 Interrupting, Continuing, and Single-Stepping

You can also interrupt execution of a program in Prism, and single-step through it. Let's run `primes` again. As before, start execution by clicking on **Run**; you don't need to reload the program. Notice that Prism reuses the same Input/Output window.

While the program is executing, click on the **Interrupt** button in the tear-off region (notice that this button doesn't turn gray; this indicates that the function is still available). When you click on **Interrupt**:

- Prism displays a message in the history region, stating where the program was interrupted.
- The source window updates to show the code being executed. The execution pointer in the line-number region points to the line currently being executed.
- The status of the program is **stopped**.

When the program is stopped, you can single-step through it:

- Click on **Step** in the tear-off region to execute the next line.
- Click on **Next** in the tear-off region to execute the next statement (a function call counts as a single statement).

In both cases, the execution pointer moves to point to the new execution point in the source window. If the step takes a long time to execute, unavailable buttons and menu selections are grayed out.

We can also issue the **stepi** and **nexti** commands from the command line to step by machine instruction. When you issue these commands, the history region displays the instruction at the address at which you stop.

Click on **Continue** in the tear-off region to continue execution without single-stepping. The program will run to completion.

3.4 Debugging a Program

In Prism, you can perform standard debugging operations like setting breakpoints and examining the call stack. Prism also provides a comprehensive method for controlling the execution of a program by means of an *event table*.

3.4.1 Setting a Simple Breakpoint

A *breakpoint* stops execution of a program. The easiest way to set a breakpoint in Prism is to left-click the mouse to the right of the line number at which you want the program to stop. Notice that when you move the mouse pointer into this area, it turns into a **B**. When you left-click next to a line, the **B** appears at that point.

Set a breakpoint at line 15 of **primes**. Note that when you do this, Prism displays a message in the history region:

```
(1) stop at "primes.fcm":15
```

Now, when you execute **primes**, the program will stop whenever it reaches line 15.

To delete the breakpoint, left-click on the **B**; it disappears.

3.4.2 Using Commands

You can set more complicated breakpoints by issuing the `stop` command on the command line. For example, issue this command:

```
stop in loop after 10
```

This tells Prism to stop execution the tenth time the program reaches the function `loop`. Note that it causes a **B** to be placed at line 32 in the line-number region, next to the first line in the routine `loop`.

Similarly, you can use the `trace` command to stop execution, then have it start again automatically.

3.4.3 Using the Event Table

The breakpoints and traces described above are *events* that control the execution of a program. Prism provides an event table that lets you create all such events in one place before you execute your program.

To display the event table, choose **Event Table** from the **Events** menu. Figure 7 shows the event table, listing the two breakpoints we discussed above.

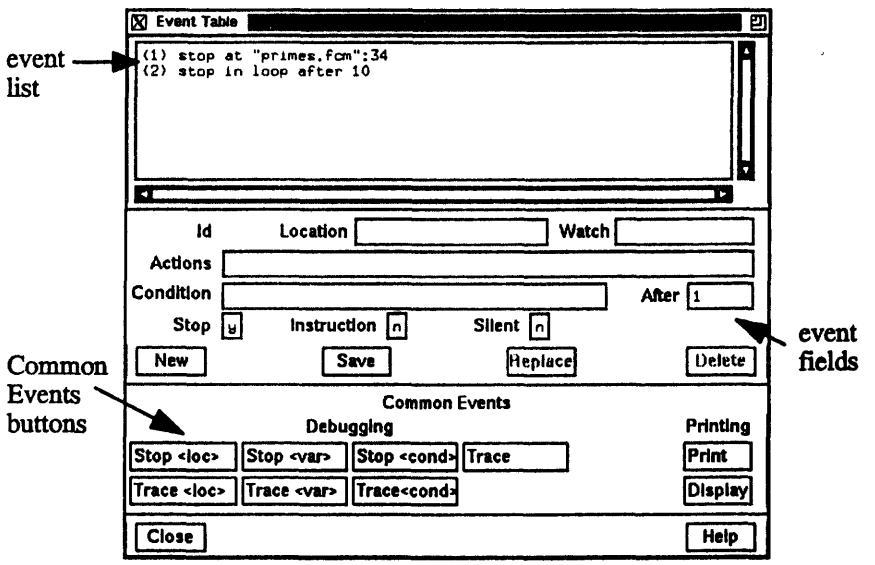

Figure 7. The event table.

The top area of the event table is the *event list* — a scrollable region in which events are listed. When you execute the program, Prism uses the events in this list to control execution. Each event is listed in a format in which you could type it as a command in the command window. It is prefaced by an ID number assigned by Prism. For example, in Figure 7, the events have been assigned the IDs 1 and 2. Creating an event by issuing a command, or in any other way, adds an event to this list.

The middle area of the event table is a series of fields that you fill in when editing or adding an event; only a subset of the fields is relevant to any one event. For example:

- Fill in the **Location** field to specify a line number when setting a breakpoint at a line.
- In the **Actions** field, specify any actions that are to accompany the event (for example, printing the value of a variable or displaying the call stack). You can include most Prism commands in this field.
- In the **Condition** field, specify a logical condition that must be met if the event is to take place.

The area headed **Common Events** contains buttons that provide shortcuts for creating certain standard events. For example, if you click on **Stop <cond>**, the **Condition** field is highlighted and the cursor is placed in it. You can then enter a condition and click on **Save** to add the event to the event list. The program will stop when the specified condition is met.

3.4.4 Displaying and Moving through the Call Stack

You can use Prism to display and move through the call stack — the list of procedures and functions currently active in a program. Choose the **Where** selection from the **Debug** menu to display a window containing the call stack; the window is updated automatically when execution stops or you issue commands that change the stack. You can click on a function in the window to make that function current. You can also click on **Up** or **Down** in the tear-off region to move up or down one level in the call stack.

If you run `primes` to the breakpoint you have set in `loop`, the call stack looks like the one shown in Figure 8. Note that the first two levels of the call stack are routines called by the CM-5 timesharing daemon and the run-time system,

respectively; you don't have to be concerned with them in debugging your program.

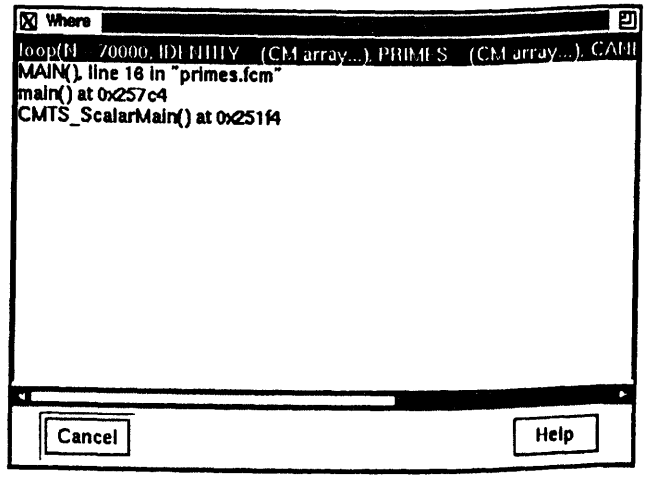

Figure 8. The Where window.

3.5 Visualizing Data

One of Prism's most important features is the ability to display graphically the values of the large arrays that are typical of data parallel programming.

In the `primes` program, the largest variable is also called `primes`; it is an array of 70,000 logicals. Each element of the array is set to *true* if its index is a prime number, and *false* if it isn't.

Let's print the values of `primes`.

First, issue the command `reload` from the command line to reload `primes`. This deletes any existing events.

Then scroll through the source window and set a breakpoint at line 62.

In the source window, find a line where `primes` appears (for example, line 43). Move the mouse pointer to `primes`. Press the left mouse button, and drag the mouse until `primes` is highlighted. Now let go of the button.

Keeping the mouse pointer in the source window, press the right button. A menu appears. Right-click on **Print** in this menu.

Prism then displays a *visualizer* for the array, as shown in Figure 9.

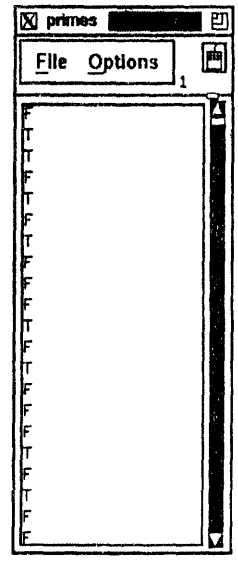

Figure 9. A visualizer for `primes`.

The figure shows the values of the first elements of the array. You can scroll through the window to see other elements.

Other representations are available besides straight text. Left-click on **Options** to display a menu. Then left-click on the **Representation** selection to display another menu. Left-click on **Threshold** in this menu. The T's and F's apparently disappear. In fact, the T's have changed into black pixels, the F's into white pixels. To make them more visible, follow these steps:

1. Left-click on **Options** again.
2. Left-click on **Parameters** from the menu.
3. Change the number in the **Field Width** box to 20.
4. Left-click on **Apply**. The black values will be much more visible.

Now left-click on **Ruler** in the **Options** menu. A ruler surrounds the values, making it easier to see separate elements. Figure 10 is an example.

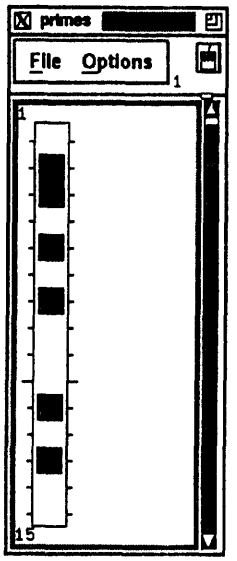

Figure 10. A threshold visualizer for `primes`.

To see the index of an individual value, move the mouse pointer onto the value. Press the **Shift** key; while pressing it, press the left mouse button. The index and value are displayed.

To close the visualizer, left-click on **File** in the visualizer window, then left-click on **Close**.

3.5.1 Other Representations

In addition to the text and threshold visualizers shown above, Prism provides a variety of other representations that you may find useful, depending on the kind of data you want to look at:

- Use the **Colormap** representation (if you have a color workstation) to display the values as a range of colors.
- Use the **Dither** representation to display the values as a shading from black to white.
- Use the **Graph** representation to display the values as a graph, with the index of each array element on the horizontal axis and its value on the vertical axis.

- Use the **Surface** representation (if your data has more than one dimension) to render the 3-dimensional contours of a 2-dimensional slice of data.
- Use the **Vector** representation to display complex numbers as vectors.

You can change the parameters of a visualizer to adjust the display. For example, you can pick a minimum and a maximum for many of these representations.

In addition, for most representations, you can set a context, using an expression that evaluates to true or false for every element of the array; the values of elements that evaluate to false are grayed out.

Finally, you can display summary statistics for an array or parallel variable, showing its minimum, maximum, mean, and other information.

3.5.2 Other Methods of Choosing the Data to Visualize

In the example at the beginning of this section, we chose the variable to visualize by dragging the mouse over it in the source window. Prism provides several other methods for choosing the variable. You can also visualize expressions.

In Prism, you can either *print* or *display* data. Printing data shows the value(s) of the data at a specified point during execution. Displaying data causes its value(s) to be updated every time the program stops execution. (You can also update a print visualizer, by choosing an option from its **File** menu.)

Here are some of the ways in which you can print or display data:

- You can create print and display events in the event table, or by choosing **Print** or **Display** from the **Events** menu. This lets you specify, for example, the location in the program at which the printing is to take place.
- You can print or display at the program's current stopping place by choosing **Print** or **Display** from the **Debug** menu. This pops up a dialog box in which you can specify the variable or expression, along with the kind of window in which you want the values to be shown.
- You can print from the source window by pressing the **Shift** key while pressing the left mouse button and dragging over a variable or expression. (If you don't press the **Shift** key, a menu pops up, from which you can choose **Print** or **Display**; that's what we did above.)

- From the command window, you can issue the **print** or **display** command. As part of the command, you can include a mask that sets the context for printing or displaying. For example,

  ```
  where (x .ne. 0) print x
  ```

  tells Prism to print the values of the array **x** in the history region of the command window; the cases in which **x** is not equal to 0 are omitted.

3.6 Obtaining Performance Data

Prism lets you collect performance data on your program. Collecting and analyzing performance data can help you uncover and correct bottlenecks that slow down a program.

Prism collects separate data for the nodes and the partition manager — referred to as the *subsystems* — and for different computing resources within each subsystem. Note that this is different from the way most other performance analyzers and profilers work; they simply measure CPU time.

To collect performance data, compile your program with the **-cmprofile** option.

If you haven't already compiled the program with this option, you can recompile from within Prism, then reload the program:

1. Choose the **Shell** selection from the **Utilities** menu. Prism creates a window containing a UNIX shell. You can issue UNIX commands from this shell, just as you would from any UNIX shell.

2. Issue this command to recompile **primes** with the **-cmprofile** option:

   ```
   % cmf -cmprofile -o primes primes.fcm
   ```

3. When the program is recompiled, you can load it from within Prism. Click on **Load** in the tear-off region. A dialog box appears. Find **primes** in the list of programs. Click on it to highlight it, and then click on **Select**. Prism then loads the program.

3.6.1 Collecting Performance Data

To collect performance data, you must turn collection on before running the program; collection then remains on until you explicitly turn it off. To turn collection on, click on **Collection** in the tear-off region. The toggle box next to it is filled in.

Then execute your program, as described in Section 3.3. (Note that you shouldn't set breakpoints or otherwise interrupt execution, since this distorts the performance data.) When the program finishes execution, the data is ready for display.

3.6.2 Displaying Performance Data

Prism displays three different levels of performance data:

- Performance statistics for the resources, along with totals for each subsystem.
- Per-procedure performance statistics for a specified resource or subsystem. You can choose either flat or call-graph display of these statistics.
- Per-source-line performance statistics for a specified resource and procedure.

All statistics are displayed as panes in the **Performance Data** window, along with the percentage or the amount of time that each histogram bar represents.

To display the performance data, choose **Display Data** in the **Performance** menu. You see a **Performance Data** window like the one shown in Figure 11.

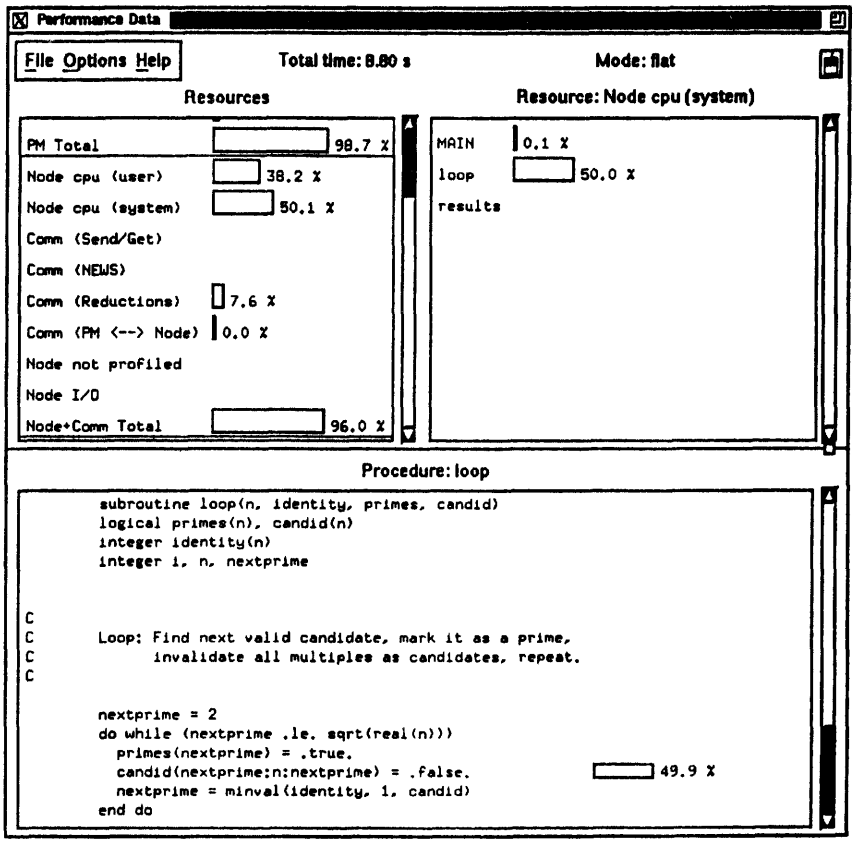

Figure 11. A Performance Data window for `primes`.

At the top left of the **Performance Data** window is a pane that shows the usage of the CM resources. Note that the partition manager (PM) and its resources are measured separately from the nodes and their resources. The resources of each subsystem can sum to 100 percent, because both the partition manager and the nodes can be active for the entire time the program is running. In the case of the `primes` program, most of the activity is on the partition manager.

To the right of this pane is a pane that shows individual procedures' use of a specific resource — in this case, the **Node cpu (system)** resource. Left-click on a resource or subsystem to display its procedure-level data.

Figure 11 shows the procedure data in *flat* mode. In flat mode, the window lists all procedures in the program and each one's total use of the selected resource or subsystem.

You can also display the procedure-level data in *call-graph* mode. In call-graph mode, you see which procedures call which other procedures, and the use of the

selected resource or subsystem for each individual call. In this case, procedure **MAIN** calls the other procedures listed; each accounts for the listed percentage of time that **MAIN** uses of the resource. To switch modes, choose **Mode** from the **Options** menu in the **Performance Data** window.

At the bottom of the **Performance Data** window is a pane that shows per-source-line usage of the selected resource or subsystem. Click on a procedure to show the source lines in that procedure.

To close the window, choose **Close** from the **File** menu.

3.6.3 Other Performance Analysis Features

Prism has additional features you can use in analyzing the performance of your program:

- Prism's *performance advisor* prints out a summary and analysis of the performance data it has collected. It zeroes in on the lines of code that are the heaviest users of the most-used resource. Working on these lines will result in the greatest performance gains in your program. To display the advisor, click on **Advice** in the **Performance** menu, or issue the **`perfadvice`** command.

- You can save your performance data to a file and subsequently reload it. This is useful if, for example, you collect the data in commands-only Prism, and you subsequently want to view it graphically. To save the data, choose **Save Data** from the **Options** menu in the **Performance Data** window or issue the **`perfsave`** command. To load the data, choose **Load Data** from the **Options** menu in the **Performance Data** window or issue the **`perfload`** command.

3.7 Getting Help

There are several ways in which you can get help in Prism:

- The **Help** menu in the menu bar provides help on several major topics. It includes the Help Index, which gives in-depth information about all aspects of Prism.

- The **Help** selection in menus and the **Help** button in windows and dialog boxes provide instructions for using these screen areas.
- Command-line help provides information about commands you can issue from the command window.

Choose **Index** from the **Help** menu to display the Help Index. Figure 12 shows the Help Index.

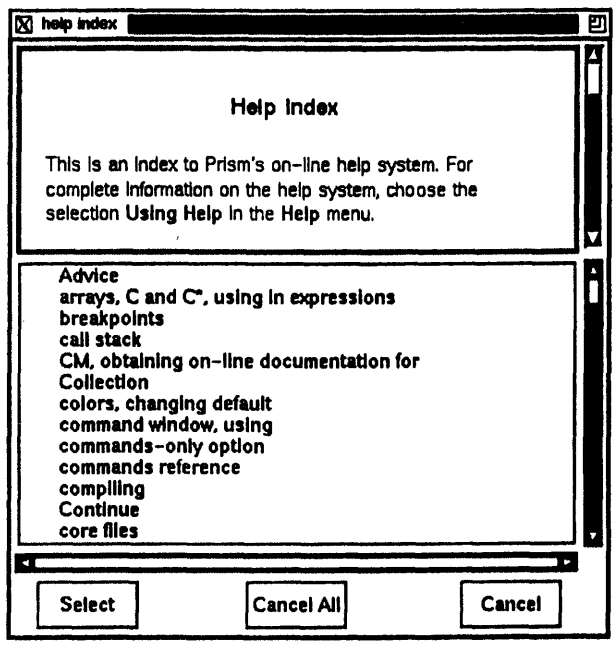

Figure 12. The Help Index.

Left-click on **call stack**, then left-click on **Select** to display the topic shown in Figure 13.

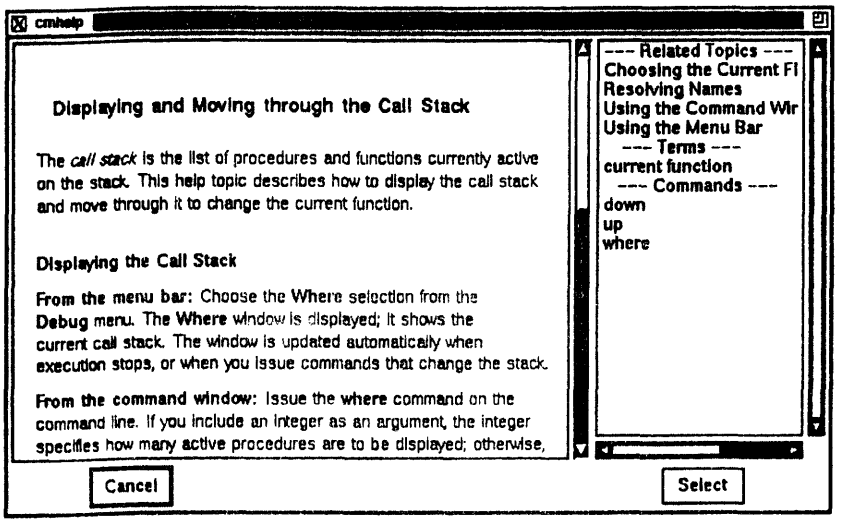

Figure 13. A help topic.

Each topic window can have lists of related topics, subtopics, terms, and commands to the right of the topic discussion. Choose an item in these lists in the same way you chose a topic from the main Help Index.

3.7.1 Other Help Features

Prism's help system contains many other useful features. For example:

- Choose **Tutorial** from the **Help** menu to display a tutorial that will guide you through loading, executing, and analyzing the `primes` sample program.
- Choose **Commands Reference** from the **Help** menu to display a list of all Prism commands. Left-click on a command, then left-click on the **Select** button to display information about it.
- Choose **Release Notes** from the **Utilities** menu to display release notes for Thinking Machines' software products.
- Choose **Man Pages** from the **Doc** menu to view an on-line copy of a manual page for a CMOST command or a language or library routine.

- Choose **Online Doc** from the **Doc** menu to view sections of on-line documentation about the CM-5. This brings up a dialog box in which you can enter the topic on which you want information. Clicking on **Search** in this dialog box passes the topic to a special version of **`xwais`**, Thinking Machines' wide-area information server; **`xwais`** searches through the documentation and lists the most relevant sections. You can then display any of these sections.

3.8 Customizing Prism

Prism provides several methods for changing the way it looks and acts:

- Choose **Tearoff** from the **Utilities** menu to enter tear-off mode. In tear-off mode, clicking on a menu selection adds a button for that selection to the tear-off region below the menu bar. Clicking on a button in the tear-off region while in tear-off mode removes the button. You can also use the **`tearoff`** and **`untearoff`** commands to do the same thing. Use the **`pushbutton`** command to add a Prism command to the tear-off region.
- Choose **Customize** from the **Utility** menu to display a window you can use to change various default Prism settings — for example, its behavior when there is an error.
- You can also add settings to your X resource database that change Prism defaults.

3.9 Leaving Prism

To leave Prism, choose **Quit** from the **File** menu. Prism displays a dialog box asking you to confirm.

3.10 Commands-Only Prism

Commands-only Prism runs from your UNIX shell; you would use it if you were logged in to a partition manager from a dumb terminal, for example, and X wasn't available to you.

Use the `-c` option to the `prism` command to start up commands-only Prism. You receive this prompt:

```
(prism)
```

You can issue any Prism command at the prompt; output appears below the command you type, instead of in the history region above the command line. There are, of course, limitations: for example, you can't create visualizers in commands-only Prism, and you cannot obtain the graphical display of the performance analysis data. You can, however, use the **`print`** or **`display`** command to print out the values of data, and you can use the **`perf`** command to display a text version of the performance data.

To obtain on-line documentation in commands-only Prism, issue the **`doc`** command.

Issuing **`doc`** displays a menu of available documents (including release notes and bug-update files). Choose the number associated with the document you want to view. In most cases, this displays another menu of the chapters within the document. Choose the number associated with the chapter, and the first screenful of text for that chapter is displayed. Answer **y** in response to the **more?** prompt or simply press the **Return** key to display the next screenful. Answer **n** to return to the menu. You can also view manual pages by typing **m** from a menu.

Issue the `quit` command to leave commands-only Prism and return to your UNIX prompt.

3.11 Using Prism with CMAX

You can use Prism's split-screen capability to debug and analyze the performance of a CM Fortran program you have created using the CMAX Converter, comparing it with the original Fortran 77 program. See Section 4.6 for more information.

Chapter 4

CM-5 Languages and Libraries

This chapter gives an overview of the languages, libraries, and related software currently available for the CM-5. For complete information, see the manuals for the individual products.

4.1 CM Fortran

Fortran for the Connection Machine system is standard Fortran 77, supplemented with the array-processing extensions of the ANSI and ISO standard Fortran 90.

The array-processing extensions provide convenient syntax and numerous intrinsic functions for manipulating arrays. For example, Fortran 90 allows an array to be treated either as a set of scalars or as a first-class object. Thus, in the statement `A = A + 1`, `A` can be a scalar, a vector, a matrix, or a higher-dimensioned array. In any case, the statement will cause all elements of `A` to be operated on in parallel. Array sections can also be specified and can be used anywhere whole arrays are used: in expressions and assignments and as arguments to procedures.

Version 2.1 of CM Fortran also supports Fortran 90 pointer arrays and 64-bit integers.

Newly written Fortran programs can use these array extensions to express efficient data parallel algorithms for the CM. These programs will also run on any other system, serial or parallel, that implements Fortran 90. CM Fortran also offers several extensions beyond Fortran 90, such as the `FORALL` statement and some additional intrinsic functions. These features are well known in the Fortran community and are particularly useful in data parallel programming.

Many existing Fortran 77 programs can be converted into CM Fortran with the assistance of the CMAX Converter, discussed in Section 4.6.

4.1.1 Programming Models

Version 2.1 of CM Fortran on the CM-5 supports both data parallel and message-passing programming styles. (Earlier versions support only the data parallel programming style.)

- When used alone, CM Fortran operates in a "global" manner, as it does on the CM-2 and CM-200. That is, it handles scalar data on the partition manager, but lays out parallel arrays across all the processing nodes of the CM-5 partition on which it executes. (CM-2/200 users should note that the compiler's view of the CM-5 hardware is identical to the slicewise execution model on the CM-2/200.) Parallel data is laid out across the processing elements, with each element executing elemental code on its local data, independently of the others. The partition manager executes scalar code and calls run-time functions for inter-processor communications.

  Within this basic pattern, two execution models are supported:

  - The "VU model" makes use of the optional vector units on the CM-5. It lays out and processes parallel arrays on all the vector units within the partition.

  - The "nodes model" ignores the vector units. It lays out parallel arrays in SPARC memory across the processing nodes, and uses the SPARC microprocessor within each node to operate on them.

  The compiler switches `-sparc` and `-vu` select the desired execution model.

- When used within a CMMD message-passing program, CM Fortran operates in a "local" manner. Independent copies of the CM Fortran program run on every node. Scalar data is handled by the microprocessor on the node; parallel data is handled by either the microprocessor or the VUs, depending on the available hardware and on the compiler switch chosen. Communication among processors is handled by the application via CMMD message-passing routines.

4.1.2 Intrinsic Functions

Fortran 90 defines a rich set of intrinsic functions that take an array object as argument and use parallel computation to construct a new array (or scalar). Intrinsics include reduction intrinsics (such as **SUM** and **MAXVAL**) and parallel prefix (or scan) operations; array construction functions such as **TRANSPOSE**, **RESHAPE**, **PACK**, **UNPACK**, and **SPREAD**; and array multiplication functions (**DOTPRODUCT** and **MATMUL**). In addition, CM Fortran offers such instrinsic functions as **DIAGONAL**, **REPLICATE**, **RANK**, **PROJECT**; bit intrinsics such as **FIRSTLOC**, **LASTLOC**, **LEADZ**, **POPCNT**, and **POPPAR**; and a set of intrinsics (new with Version 2.1) that provide a form of "equivalence" (or storage association) on array subgrids.

4.1.3 Utility Library

The CM Fortran Utility Library provides a convenient interface to CM-5 operations that the language cannot express easily or that the compiler does not yet generate. The Utility Library provides an interface from CM Fortran to lower-level software such as run-time functions. It also provides the CM Fortran interface to I/O. Utility Library I/O calls can access any CM file system.

4.1.4 Development and Monitoring Facilities

CM Fortran programs are typically developed within the Prism programming environment, with its graphical debugger, performance analysis tools, and data visualizers.

Timing of CM Fortran programs is accomplished using the CMOST timing functions, such as `CM_timer_start`, `CM_timer_print`, and so on.

Run-time safety is enabled by the `cmf` compiler switch `-safety` or (for a subset of safety checks) `-argument_checking`.

4.1.5 Documentation Provided

- *Getting Started in CM Fortran*
- *CM Fortran Programming Guide*

- *CM Fortran Reference Manual*
- *CM Fortran Utility Library Reference Manual*
- *CM Fortran User's Guide*
- *CM Fortran Optimization Notes: Slicewise Model*
- *CM Fortran Array Operations Quick Reference Guide*
- On-line manual pages for all CM Fortran intrinsic functions and Utility Library procedures, and release notes for the version of CM Fortran you are using

4.2 C*

C* is an extension of the C programming language, designed to support data parallel programming. It is based on the standard version of C specified by the American National Standards Institute (ANSI).

C* extends C with a small set of new features that allow programmers to use the Connection Machine system efficiently. Examples are **shapes**, which are used to logically configure parallel data; parallel versions of arrays and structures; and **where** statements, which restrict the set of positions within an array on which operations are to take place.

C* also adds a few new operators to Standard C. For example, the `<?` and `>?` operators are available to obtain the minimum and maximum of two variables (either scalar or parallel). The corresponding compound assignment operators `<?=` and `>?=` are also provided. The operator `%%` provides a true modulus operation (as compared to the remainder operator `%`).

In addition, C* provides parallel functions for computation and communication. Functions may be overloaded: you can declare more than one version of a function with the same name (one for scalar data, for example, and another for parallel data). The compiler automatically chooses the right version.

The C* language on the CM-5 is identical to C* on the CM-2 and CM-200. There are, however, some implementation differences of which programmers should be

aware. These differences are detailed in the *C* Release Notes* for Version 7.1. The most important ones are as follows:

- The CM-2/200 C* restrictions on shape extents are not present in CM-5 C*. The only restriction is that the size of each dimension must be greater than 0. (Note that on a CM-5 with vector units, the size of the physical shape is the number of vector units in the partition, not the number of nodes.)
- On the CM-5, parallel `bools` occupy 1 byte of storage, not 1 bit, as on the CM-2 and CM-200. The semantics of using `bools` remain the same; you need not change an existing program to deal with the new size. Memory usage will go up on the CM-5, however.
- C* on the CM-5 supports parallel enums.
- Because the CM-5 C* compiler is generally compliant with the ANSI standard, it will reject some programs that previously compiled without error.
- CM-5 C* programs cannot call Paris routines.

In addition, the `cs` command has several new options for use on the CM-5, but does not accept certain CM-2/200 options. For example, the **-vu** option specifies compilation for a CM-5 with vector units; the **-sparc** option specifies compilation for a CM-5 without vector units.

Version 7.1 of C* supports node-level (message-passing) programming, in the same manner that CM Fortran Version 2.1 does.

4.2.1 Documentation Provided

- *Getting Started in C**
- *C* Programming Guide*
- *CM-5 C* User's Guide*
- *CM-5 C* Performance Guide*
- Release notes for the version of C* that you are using

4.3 CM Scientific Software Library

The Connection Machine Scientific Software Library (CMSSL) is a growing set of numerical routines that support computational applications while exploiting the massive parallelism of the Connection Machine system. CMSSL provides data parallel implementations of familiar numerical routines in the areas of linear algebra, ordinary differential equations, signal processing, statistical analysis, and linear programming. It also offers a number of communication functions that facilitate computations on both structured and unstructured grids.

On the CM-5, CMSSL is callable from CM Fortran. The user interface for these routines is identical with the CM Fortran user interface offered on the CM-200; the actual implementation of the routines, however, often differs, to take best advantage of the hardware of each machine.

Version 3.1 of CMSSL concentrates on six critical areas of programming: numerical linear algebra, Fourier Transforms, ordinary differential equations, linear programming, random number generation, and statistical analysis.

4.3.1 Linear Algebra

Most CMSSL linear algebra routines are designed to support multiple instances. The difference between invoking computation on a single instance and on multiple instances lies only in the dimensionality and layout of the data structures used as parameters to the CMSSL routine.

Within the general area of linear algebra, CMSSL offers:

- Matrix operations on dense, grid sparse, and arbitrary sparse matrices. For dense matrices, CMSSL includes inner and outer product routines; matrix, matrix vector, and vector matrix multiplication routines; a 2-norm routine; and an infinity norm routine. For grid and arbitrary sparse matrices, the library provides matrix, matrix vector, and vector matrix multiplication.

- Linear equation solvers for dense, banded, and sparse systems of equations: *LU* and *QR* factorization and solution routines, triangular system solvers, a Gauss-Jordan system solver, and matrix inversion; factorization and solution of banded systems via pipelined Gaussian elimination (with optional pairwise pivoting) or via substructuring with either cyclic reduction, balanced cyclic reduction, pipelined Gaussian elimination, or transpose; and several standard iterative solvers, including the Conjugate

Gradient, Bi-Conjugate Gradient with Stabilization, Quasi-Minimal Residual, and Restarted Generalized Minimal Residual methods.

- Eigensystem analysis of real symmetric tridiagonal, dense Hermitian, dense real symmetric, and sparse systems, via a number of methods including Jacobi rotations, *k*-step Lanczos method, and *k*-step Arnoldi method.

4.3.2 FFTs

CMSSL offers routines for the computation of Fourier Transforms by Cooley-Tukey type algorithms on one or more axes of arrays with an arbitrary number of axes. Currently, a complex-to-complex FFT is provided. Real-to-complex and complex-to-real FFTs are planned for a future release.

4.3.3 Ordinary Differential Equations

CMSSL provides a routine that solves the initial value problem for a system of *N*-coupled first-order ordinary differential equations by explicitly integrating the set of equations using a fifth-order Runge-Kutta-Fehlberg formula.

4.3.4 Linear Programming

CMSSL provides a routine that solves multidimensional minimization problems using the simplex linear programming method. The goal is to find the minimum of a linear function of multiple independent variables.

4.3.5 Random Number Generation

CMSSL provides two random number generators. Both use a lagged-Fibonacci algorithm to produce a uniform distribution of random values. Both may be reinitialized for reproducible results.

4.3.6 Statistical Analysis

CMSSL offers two histogramming operations: one that tallies the occurrences of each value in a CM array, and one that counts the occurrences of values within specified value ranges. The latter facilitates breaking data from particularly large data sets into subranges, perhaps as a preliminary step before doing more detailed analysis of interesting areas.

4.3.7 Communication Functions

CMSSL includes routines for efficient data motion for nearest-neighbor operations on regular grids, for all-to-all communication on segmented arrays, and for gather and scatter operations on unstructured grids. The library also provides utilities for data distribution for load balancing of communication. Routines offered include polyshift, all-to-all broadcast, several gather and scatter utilities, and partitioning of an unstructured mesh. There is also a communication compiler, a set of routines that compute and use message delivery optimizations for basic data motion and combining operations. The communication compiler allows you to compute an optimization (or trace) just once, and then use the trace many times in subsequent data motion and combining operations.

4.4 Visualization Programming

4.4.1 A Distributed Graphics Strategy

In keeping with its role as a network resource, the CM-5 uses a distributed graphics strategy to support a wide range of user applications. The key items in this strategy are;

- the parallel processing power of the Connection Machine supercomputer
- the specialized power and interactive visualization environments, such as AVS, provided by dedicated graphics display stations
- the use of standard protocols, such as X11, to allow communication among a variety of hardware and software

A full range of interconnections is supported, from high-speed HIPPI interfaces through FDDI and Ethernet for longer-distance communications, to allow fast communication between the CM and graphics display stations.

As an example, a scientific visualization program could use the CM to compute image geometry (including, for example, polygon coordinates and color information) and then send it from the CM directly to local memory on the graphics workstation, where the results of simulations done on the CM can be displayed and analyzed interactively.

4.4.2 An Integrated Environment

By using the distributed graphics strategy described above, together with an underlying protocol such as X11 or the AVS graphical user interface, programmers can create and use a wide variety of integrated environments for their computational and visualization tasks. Connection Machine software provides an environment that permits the exchange of very large data sets between the CM and framebuffers, workstations, or X window terminals.

4.4.3 The CM/AVS Visualization Environment

CM/AVS adapts and extends the Application Visualization System (AVS, from Advanced Visualization Systems, Inc.) to the realm of the CM-5. This graphical user interface enables an application to operate on data that is distributed on CM-5 processing nodes and to interoperate with data from other sources. A user normally runs AVS on a local workstation and uses the modules and functions that CM/AVS provides to process data on the CM-5. That way, the advantages of user-interface-intensive workstation visualization are combined with the power of data-intensive CM-5 applications.

The building blocks of an AVS application program are small, packaged units of code, called *modules*. Most modules process a set of inputs into a set of outputs. Eah module incorporates a function, which can be as simple as adding two arrays or as complicated as rendering the isosurfaces of a volume. AVS modules execute on the workstation; CM/AVS modules execute on the CM-5. Hundreds of visualization modules are available from AVS and Thinking Machines and in the public domain.

Data for module inputs and outputs is typed. CM/AVS provides a parallel version of the AVS "field" data type used to represent arbitrary arrays of data. CM/AVS's parallel field data is allocated on the CM-5 processing nodes as CM Fortran arrays or C* parallel variables.

Within CM/AVS, parallel fields appear identical to regular serial fields; the two may be used interchangeably. When CM/AVS modules that operate on parallel data are connected with AVS modules that operate on serial data, CM/AVS routines convert the data between parallel and serial fields as required. The conversion is transparent to the user and to the module writer.

Documentation Provided

- *CM/AVS User's Guide*
- *CM/AVS Release Notes*
- On-line man pages viewable through the AVS man page viewer, and release notes for the version of CM/AVS you are using

In addition, users should have the AVS document set.

4.4.4 Visualization Programming with CMX11

The CMX11 visualization library is designed for distributed graphics programming in a heterogeneous computing environment. This library, callable from CM Fortran and C*, allows you to display data from CM-5 memory on an X windows server screen anywhere on your network.

Creating and Controlling a Display

The CMX11 library provides functional CM Fortran and C* interfaces that make it easy to create and control one or more windows on an X11 server as a CM display.

- Specify the X11 server you wish to use by setting the environmental variable `DISPLAY` or by using the `-display` option on the command line when you invoke your program.
- Within your program, call the subroutine `CMXCreateSimpleDisplay` to connect to the display specified.

Other routines allow you to manage the display from your application. For example, you can get information on the display size or set the display colors.

Rendering Your Data

The CMX11 library provides parallel extensions to the standard X drawing primitives that accept parallel arrays of coordinate and color information. These routines enable you to draw large sets of points, lines, arcs, rectangles, filled polygons, text strings, or an image array — each with a single subroutine call.

Graphics Programming

The basic CMX11 drawing and display capabilities do not require any X programming. However, the library provides routines that give you access to the underlying X structures. If you are an experienced X programmer, these enable you to integrate your CMX11 program with an existing X or Motif application.

Documentation Provided

Anyone who wants to perform simple visualization operations on the results of CM-5 computations will find sufficient information in the Thinking Machines CMX11 documentation:

- *CMX11 Reference Manual*
- *CMX11 Release Notes*
- On-line man pages for all CMX11 routines, and release notes for the version of CMX11 you are using

For more elaborate graphics programming, users who are unfamiliar with X may wish to consult the following publications in addition:

- *Xlib Programming Manual,* Adrian Nye (Sebastopol, CA: O'Reilly, 1988)
- *Xlib Reference Manual,* Adrian Nye (Sebastopol, CA: O'Reilly, 1990)
- *X Window System User's Guide,* Valerie Quercia and Tim O'Reilly (Sebastopol, CA: O'Reilly, 1990)

4.5 Message Passing with CMMD

The CMMD communications library supports message-passing programming on the CM-5. This programming involves explicit message passing between processing nodes.

The CMMD library is callable from C, C++, and Fortran 77; programs are compiled with the appropriate Sun compiler. At Version 3.0, CMMD is also callable from CM Fortran Version 2.1, with the program compiled by the `cmf` compiler, and from C* Version 7.1, with the program compiled by the `cs` compiler. Programs or program modules written in CM Fortran or C* use the vector units on the CM-5. Other programs or modules use only the microprocessor on each node.

4.5.1 Programming Models

CMMD supports two programming models:

- The host/node programming model involves writing two programs that will run simultaneously. One program runs on the host, while an independent copy of a second program runs on each processing node. On the CM-5, the host is the partition manager that controls a given partition, while the nodes are the processors within the partition. The host begins execution by performing needed initializations (including initializing the CMMD message-passing environment) and then invoking the node program; it may have little involvement in subsequent computations.

- The hostless programming model uses the host only to initiate execution and to act as an I/O server. A CMMD-supplied host program performs these tasks; the user writes a single application, which runs on each of the nodes. The nodes pass messages to each other, but do not explicitly talk to the host.

4.5.2 Cooperative Processing and Asynchronous Processing

CMMD supports both cooperative and asynchronous message passing. With cooperative concurrent message passing, synchronization occurs only between matched sending and receiving nodes and only during the act of communication. At all other times, computing on each node proceeds asynchronously with respect to the other nodes.

A set of global functions provides for broadcast, reduce, scan, and concatenate operations, and for global synchronization. As their name implies, global functions involve all nodes in the partition; executing the function synchronizes the nodes.

CMMD also permits fully asynchronous message passing. Asynchronous message passing is usually interrupt-driven: a node signals a readiness to send or receive a message, then performs other work until its partner node is ready for the transmission. If preferred, however, asynchronous message passing may be driven by polling.

To optimize performance of repeated patterns of message passing, CMMD provides virtual channels. With channels, two nodes establish a one-way transmission link that can be used multiple times. Once the channel is established, the sending node writes a predefined array into the channel; the receiving node reads the channel, then resets it for another use. No synchronization is needed.

4.5.3 Remote Memory Access and Active Messages

In addition to the message-passing functionality listed above, which uses a handshake protocol, CMMD provides protocol-free messaging capabilities. These are particularly useful for programmers who want to define their own protocols. Two major items in this area are remote memory access and the Active Message facility.

- With remote memory access, one node reads or writes a portion of a second node's memory, as if the two nodes shared a common memory.
- With the Active Message facility, one node activates some routine on a second node. The routine may itself activate further routines that either respond to the first node or activate routines on yet other nodes.

4.5.4 CMMD I/O

CMMD extends UNIX I/O to provide for both independent and cooperative I/O. A file may be open for a single node or for all nodes. If open for all nodes, it can be in one of three modes:

- In independent mode, any node can read or write the file independently.

- In synchronous-sequential mode, all nodes read and write the file simultaneously, each reading or writing a separate, but sequential, portion of the file.
- In synchronous-broadcast mode, one portion of a file is read and broadcast to all nodes simultaneously.

CMMD also provides double-precision file pointers, to allow applications to access very large files.

4.5.5 Supporting Utilities

The supporting facilities provided for data parallel programs, such as the CM timers, typically treat the nodes as a collective, since the nodes each store part of the same data set. Message-passing programs, in contrast, are supported by facilities that allow independent access to each node: for example, the CMMD timers and the **pndbx** debugger (used either independently or from within Prism). The *CMMD User's Guide* provides hints for using program development and monitoring facilities.

4.5.6 Documentation Provided

- *CMMD Reference Manual*
- *CMMD User's Guide*
- On-line man pages for all CMMD routines, and release notes for the version of CMMD you are using

4.6 The CMAX Converter

CMAX — the "CM Automated X-lator" — is an aid to converting standard Fortran 77 into CM Fortran. CMAX provides a convenient migration path for serial programs onto the massively parallel Connection Machine system, both for data parallel applications and for CM Fortran/CMMD message-passing applications.

computer environment and third-party software developers can use the converter as a "preprocessor" for routine Fortran compilation for CM systems. In this sense, CMAX provides a migration path onto the Connection Machine system.

The major difference between serial and data parallel Fortran programs is the substitution of array operations for loop iterations, and the concomitant need to lay out some arrays across the processing nodes. These are the tasks performed by the CMAX converter.

CMAX is a DO loop vectorizer. It analyzes loop constructs and translates them into CM Fortran array operations. For greatest efficacy, the converter performs an interprocedural dependence analysis of the whole program (not just of individual subroutines) and applies vectorization techniques such as loop fissioning, scalar promotion, and loop pushing to the input code. CMAX also recognizes the intent of numerous programming idioms, such as structured data interactions and dynamic array allocation. When translating code, it makes full use of powerful Fortran 90 features such as array-processing intrinsic functions and dynamic allocation statements, as well as the FORALL statement defined by High Performance Fortran. CMAX thus provides entree both to the Connection Machine system and to the emerging HPF standard.

CMAX provides a convenient interface to the user. The Prism development environment provides facilities for examining CMAX output and comparing it line-by-line with the input program; see Figure 14, which shows a Prism window with the CM Fortran code in one pane, and the corresponding Fortran 77 code in the other pane. CMAX command options and in-line directives allow the user to control the converter's actions and decision rules. The CMAX library provides canonical, portable — and translatable — Fortran 77 utilities for expressing common operations like dynamic array allocation and circular array element shifts. The converter generates detailed notes of a conversion, explaining all the changes it has made.

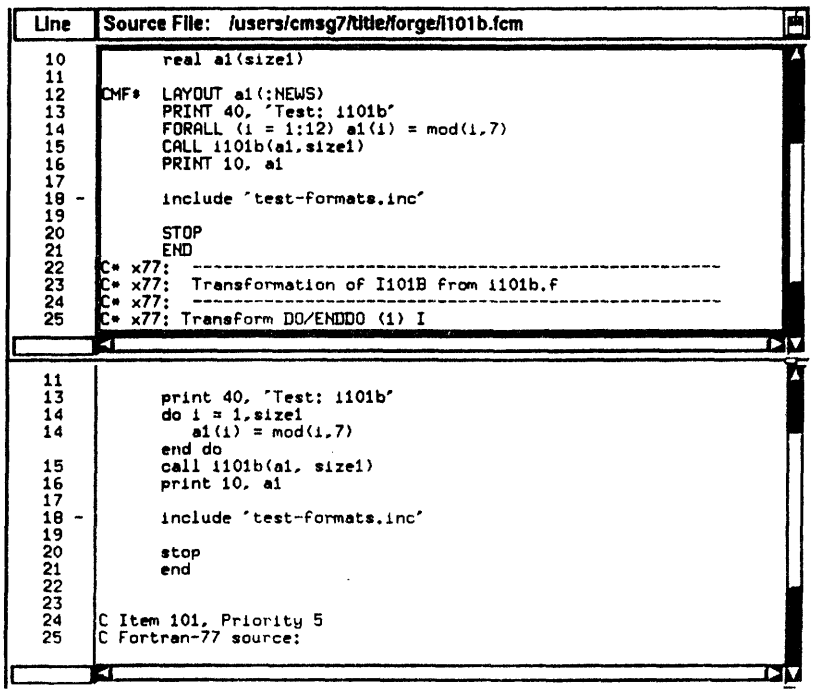

Figure 14. CM Fortran and Fortran 77 code in a split screen.

Although CMAX is designed primarily to assist in the creation of new applications, it accepts as input any program that is written in standard Fortran 77 and follows standard guidelines for scalability. These simple guidelines guarantee that a program runs efficiently on any size data set, large or small, and on any number of processors, from one to thousands. The combination of guidelines plus converter can assist substantially the task of upgrading "dusty deck" programs to take advantage of modern architectures and language features.

The conventions of scalable Fortran programming express three basic objectives:

- Make it easy for a compiler to recognize how data and computations may be split up for independent or coordinated processing. For example: loop over as many array axes as possible in a single operation; use standard idioms to express common, well-structured data dependences.
- Avoid constructions that rely on a particular memory organization, such as linearizing multidimensional arrays or changing array size or shape across program boundaries.

- Use data layout directives and library procedures (with some conditionalizing convention) to take advantage of the specific performance characteristics of each target platform. For example, Fortran 77 programs targeted to the CM system can use compiler directives to fine-tune data layout and access the CM libraries for procedures that are specially tuned for performance on the CM system.

4.6.1 Documentation Provided

- *Using the CMAX Converter*

4.7 Assembly Language

The initial implementation of CM-5's processing node uses a SPARC microprocessor. Low-level programming can therefore be carried out in SPARC assembly language; this is not recommended, however, since the implementation of the CM-5 processing node is expected to track the RISC microprocessor technology curve to provide the best possible functionality and performance at any given time. Thus, SPARC assembly language programs may not be portable to future CM-5s.

Thinking Machines provides an instruction set called DPEAC for programming the optional vector-unit hardware. DPEAC is an extension of SPARC assembly language, providing additional assembly-level operations that are CM-5 specific.

4.7.1 Documentation Provided

- *DPEAC Reference Manual*

Appendix A

Moving from the CM-2 to the CM-5

This appendix is for CM-2 and CM-200 users who want to port their programs to the CM-5. For complete information on porting a CM Fortran program, see the *CM Fortran User's Guide*. For complete information on porting a C* program, see the *CM-5 C* Release Notes.*

A.1 Updating the Program

A.1.1 CM Fortran

Programs written in the CM Fortran language run on both the CM-5 and the CM-2/200. You need not make any changes in the use of language features to port a program from one platform to another. Some other system features, however, are platform-dependent. For example, send addresses on the CM-5 are 8-byte integers. Arrays that contain send addresses, therefore, should be declared (on any CM) as `DOUBLE PRECISION` or `REAL*8`. Send address arrays declared in this way are portable across all CM platforms. Also, some compiler switches apply to only one platform. Optimizations are the same for the CM-5 and the slicewise model on the CM-2 and CM-200; Paris optimizations do not apply to the CM-5.

Assemblers also differ: the Paris instruction set is not supported on the CM-5, so calls to Paris must be removed from CM-2/200 programs before they are ported to the CM-5. Such calls are typically replaced by calls to the Utility Library. Other library calls may also be non-portable:

- CMMD and CMX11 are supported only on the CM-5.
- CM-2/200 visualization libraries are supported only on the CM-2 and CM-200.

- The CM Fortran Utility Library, CMSSL, and CM/AVS are supported on all CM systems.

A.1.2 C*

Most CM-2/200 C* programs should port without difficulty to the CM-5. You must recompile and relink using the CM-5 C* compiler. This list summarizes the changes that you must make (when applicable) to ensure portability:

- Remove all calls to libraries (like Paris) not supported on the CM-5.
- Remove all include files not supported on the CM-5 (for example, **<cm/paris.h>**).
- If you express lengths in terms of bits in a function (for example, in the overloaded versions of the grid communication functions or the **get** or **send** function), rewrite the code to express the size with **boolsizeof** and the appropriate parallel type.
- Change calls to **allocate_detailed_shape** to use the new format for CM-5 C*.
- The CM-5 C* compiler disallows casts between scalar types and pointers to parallel variables. If you call **palloc()** in a CM-2/200 C* program without including **<stdlib.h>** (which properly declares its return type) and cast the result, the code won't compile on the CM-5. Thus, this code won't work:

```
/* No included stdlib.h file */

int:current *p = (int:current *)palloc(current,
                  boolsizeof(int:current));
```

Change it to this so that it will work in CM-5 C*:

```
#include <stdlib.h>

int:current *p = palloc(current,
                        boolsizeof(int:current));
```

A.2 Compiling and Linking

A.2.1 CM Fortran

In compiling CM Fortran programs, note that the CM-2/200 options **-paris** and **-slicewise** are not supported on the CM-5; the CM-5 supports only the options **-sparc** and **-vu**.

On the CM-5, **cmf** does not invoke the linker **ld** directly. If you link as a separate step, we recommend reinvoking **cmf**.

A.2.2 C*

You compile and link CM-5 C* programs as you do CM-2/200 C* programs. CM-5 C* supports most CM-2/200 compiler options. Exceptions are:

- **-noline**
- **-release**
- **-ucode**
- **-pg**
- **-keep c**, since the compiler does not generate C code

A.3 Executing

When executing a CM Fortran or C* program, note that you do not issue the commands **cmattach** or **cmcoldboot**. A CM-5 partition manager is always "attached" to the nodes it controls. In addition, the **cmsetsafety** command is not available on the CM-5.

The CM-2/200 checkpointing facility is not currently supported on the CM-5.

A.4 *Lisp

CM-5 supports the *Lisp interpreter. Most CM-2/200 *Lisp code will port without any changes. However, operations that are not part of *Lisp, such as calls to Paris or other low-level facilities, either will not work on the CM-5 or will require significant revision. For complete information, see the manual *Porting to CM-5 *Lisp*.

Appendix B

A Sample CM Fortran Program

This is the source code for the CM Fortran program **primes.fcm**, used as an example in this manual:

```
      program findprimes
      implicit none
      integer i, n, nextprime
      parameter (n = 70000)
      logical primes(n), candid(n)
      integer identity(n)

C
C     Initialization
C

      identity = [1:n]
      primes = .false.
      candid = .true.
      candid(1) = .false.
      call loop(n, identity, primes, candid)
      call results(n, primes)
      end

      subroutine loop(n, identity, primes, candid)
      logical primes(n), candid(n)
      integer identity(n)
      integer i, n, nextprime

C
C     Loop: Find next valid candidate, mark it as a prime,
C           invalidate all multiples as candidates, repeat.
C

      nextprime = 2
      do while (nextprime .le. sqrt(real(n)))
        primes(nextprime) = .true.
```

```
          candid(nextprime:n:nextprime) = .false.
          nextprime = minval(identity, 1, candid)
        end do

C
C       At this point, all valid candidates are prime
C

        primes(nextprime:n) = candid(nextprime:n)
        end

        subroutine results(n, primes)
        logical primes(n)
        integer i, n
C
C       Print results
C

        print *, "Number of primes: ", count(primes)
        do i = n,1, -1
          if (primes(i)) then
            print *,i
            goto 10
          end if
        end do
  10    end
```

Appendix C

Glossary

This glossary presents brief explanations of UNIX and CM-5 terms used in this manual. For a more comprehensive discussion of the UNIX system, consult *The UNIX Programming Environment*, by Brian W. Kernighan and Rob Pike (Prentice-Hall, 1984), or one of the many other books written about UNIX.

C*	A data parallel extension of the C programming language.
CM/AVS	A graphical user interface that adapts and extends the Application Visualization System to the CM-5.
CMAX	A software tool that translates Fortran 77 programs into CM Fortran.
CM Fortran	An implementation of the Fortran 77 programming language, extended with array-processing facilities from Fortran 90.
CMFS	Connection Machine File System. A UNIX-like file system that can reside on CMIO-bus data-storage devices, such as a DataVault.
CMIO bus	An I/O bus that connects CM-2 and CM-200 I/O devices to the CM-5's Data Network.
CMMD	A communication library used in creating node-level message-passing programs on the CM-5.

CMOST	The CM-5's operating system, an enhanced version of UNIX.
CMSSL	CM Scientific Software Library. A library of routines that perform data parallel versions of standard mathematical operations.
CMX11	A library of routines that manage the transfer of parallel data between the CM-5 and any X11 terminal or workstation.
control processor	A CM-5 processor that manages partitions or I/O.
Control Network	A communication network on the CM-5, used for operations that involve all the nodes at once, such as synchronization operations and broadcasting.
`.cshrc`	In the C shell, a script file run after login to set up the characteristics of the shell.
C shell	See *shell.*
current directory	See *directory.*
Data Network	A communication network on the CM-5, used for bulk data transfers where each item has a single source and destination.
DataVault	A high-performance, disk-based mass storage system for use in CM systems.
directory	A node in the UNIX file system. A directory can contain files and other directories. The *current* or *working* directory is the directory to which relative pathnames refer.

environment variables
: Variables whose settings are available both to a shell and to programs called from within the shell. You can change the settings of these variables to provide information about your environment to programs. Compare *shell variables*.

filename
: The name of a UNIX file. See also *pathname*.

group ID
: The name of a class of users to which a user is assigned.

hostname
: The name assigned to a computer running the UNIX system.

login ID
: The name by which a user is known to the system.

make utility
: A utility that provides a mechanism for maintaining programs by ensuring that the files constituting a program all exist and are up-to-date.

node
: On the CM-5, the unit containing a microprocessor, a bus, memory, a network interface, and optionally vector units.

partition
: A group of CM-5 processing nodes, under the control of a control processor, used for executing user tasks.

partition manager
: A CM-5 control processor that supervises the nodes in a single partition.

pathname
: A name that includes all the directories that have to be traversed to reach a given file or directory. An *absolute* pathname starts with root — that is, at the beginning of the file system hierarchy: for example, `/usr/bin`. A *relative* pathname starts with the working directory: for example, `my_subdirectory/file_name`.

pndbx A debugger that provides independent access to each node, used in debugging CMMD programs.

Prism The CM-5's programming environment.

process An instance of a running program. Each process in a system has a unique process ID.

prompt A symbol that indicates that the system is ready to accept commands. You can use a shell variable to set what your prompt will be. In this guide, the prompt is displayed as a percent sign (%).

relative pathname See *pathname*.

remote operations
Commands that involve interaction with UNIX systems other than the local system to which you are logged in. The `rlogin` command allows you to log in to a remote UNIX system; the `rsh` command allows you to execute a UNIX command on a remote system without logging in; and the `rcp` command allows you to copy a file to or from a remote system.

`rlogin` See *remote operations*.

root The beginning directory in the hierarchy of the UNIX file system — specified as /.

`rsh` See *remote operations*.

script file A file that contains commands or programs to be executed. You can submit a script file for execution by NQS. Also called *shell script*.

SDA Scalable Disk Array. A high-performance, highly expandable disk storage system packaged within CM-5 cabinetry.

SFS — Scalable File System. A file system that manages the files stored on a Scalable Disk Array or Integrated Tape System.

setenv — The C shell command for setting an environment variable.

shell — A command interpreter that lets you issue commands to be executed by the kernel. There are different shells that provide slightly different features. The C shell, the Bourne shell, and the Korn shell are popular UNIX shells.

shell script — See *script file.*

shell variables — A set of predefined variables whose values you can change to customize your shell. For example, you can set the **prompt** variable to change your UNIX prompt. Compare *environment variables.*

standard input, output, and error — *Standard input* is the input device for commands. *Standard output* is the device to which commands send their results. *Standard error* is the device to which commands send error messages. Typically, all three are defined to be your terminal. You can change this — for example, by using redirection to send output to a file instead of to your terminal.

subshell — See *shell.*

superuser — A special user on a UNIX system who can read or modify any file in the system.

symbolic link — An entry in a directory that points to an already existing file with a different path. This allows a user to gain access to a file without specifying an absolute pathname.

user ID — A number associated by the system with a login ID.

vector units On the CM-5, optional high-performance arithmetic hardware. Each node has four vector units, if they are present in the system.

working directory See *directory*.

Index