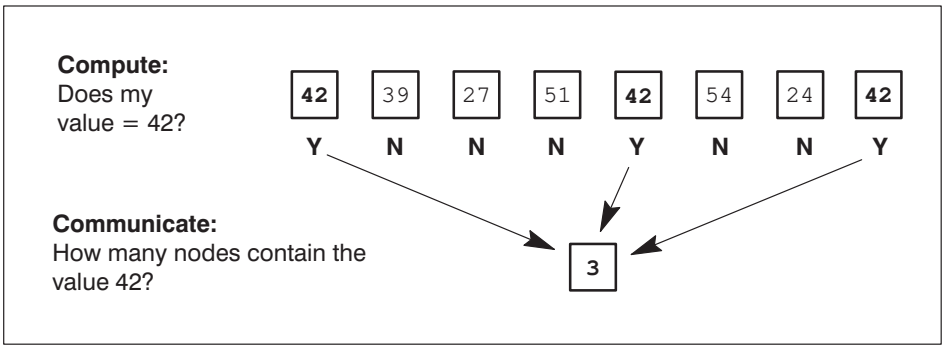
Chapter 1

Parallel Programming

A *parallel program* is one that runs on multiple processors at once. One kind of parallel program is a *data parallel* program. In a data parallel program, data in the form of *parallel variables* is distributed across all processors involved, one datum per processor. For example, an instruction to sum the squares of a parallel variable `x` would cause the squares to be computed individually on each processor; interprocessor communication would then sum the results. Data parallel execution may be *conditional* on the data values at each processor. For example, an instruction might compute the sum of the squares of parallel variable `x` only on those processors that have positive values for parallel variable `y`. Data parallel programs execute in *lock-step*, performing the same step on all processors at the same time, except for each processor's option to conditionally abstain from executing a given step.

More generally, each physical processor may simulate a number of *virtual processors*. From the standpoint of the data parallel program, virtual processors behave as though they were physical processors. When there are many virtual processors (hundreds, thousands, or more), data parallel programming is said to be *fine-grained*.



Copyright © 1995 Thinking Machines Corporation

In contrast with data parallel programs, *task parallel* programs are not lock-step or fine-grained. A task parallel program runs simultaneously on all physical processors (there are no virtual processors). All computation is performed on the data local to each processor, except when there is explicit interprocessor communication. Processors need not be doing the same thing at the same time, except as provided for by explicit interprocessor synchronization. Communication and synchronization among processors is typically done using *message passing*. The CM-5's library of message-passing routines is called *CMMD*.

On the CM-5, task parallel programs are often termed *single-program, multiple-data,* or SPMD programs. This term is meant to convey the idea that the user writes one program that is replicated on all the nodes, but all the nodes run the program asynchronously and independently. In contrast, *multiple-instruction, multiple-data* (MIMD) programming involves running a distinct program on each node. But the distinction blurs because a single program can be written to execute an entirely different module on each node.

The task parallel model is more general than the data parallel model. To use data parallelism effectively, you must decompose your problem into a number of components that can be processed similarly and simultaneously. This decomposition is not always feasible. Task parallel processing permits other decompositions: for instance, carrying out distinct, independent phases of a computation, perhaps pipelining a stream of data through successive such phases.

Task parallel programs can also be written in data parallel style, with all processors performing the same computation on successive data elements local to each processor. But the programmer may then need to attend to a number of problems that would be handled automatically by a compiler for a true data parallel program: for example, iterating over successive data elements on each processor (instead of using virtual processors, which do so implicitly); or arranging the data into some topology (for example, a two-dimensional grid) and then performing a computation between neighboring elements — which for some elements involves interprocessor communication — all of which may also be automatic with a data parallel compiler. Furthermore, task parallel programs are often more difficult to debug because there are multiple, asynchronous flows of control, instead of the single lock-step flow of data parallel execution. On the other hand, one may be able to generate more efficient task parallel code than the compiler produces for the corresponding data parallel program.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD directly supports task parallel programming, but not data parallel programming.

1.1 Parallel Programming on the CM-5

From an application's point of view, a CM-5 (or CM-5E) system includes both hardware and software. The hardware consists of

- A group of parallel processors, known as a *partition.* A partition may include all the processing nodes in the CM, or only some of them.

  On most CMs, each node contains a SPARC or SuperSPARC microprocessor, plus four vector units (VUs). The VUs create additional parallelism and accelerate floating-point arithmetic. Other CM-5s have SPARC-only nodes. In either case, each node has its own memory (either 32 or 128 Mbytes).

- A *partition manager (PM)*. This is a control processor that manages the partition. The PM initiates execution of all applications that run on the partition, and supplies them with OS services, including I/O.

- Two interprocessor networks, the *Data Network* and the *Control Network.* The Data Network transmits messages between individual nodes, between nodes and PM, and between nodes and I/O devices. The Control Network performs global operations, such as broadcasts.

- All *I/O devices* that are either part of or connected to the CM.

- Any *control processors* that control those I/O devices.

Software on the CM-5 includes both standard UNIX software that runs on the partition manager and CM system software such as

- CMOST, the CM-5's UNIX-based timesharing operating system, along with utilities such as the Distributed Job Manager (DJM).

- The CM's own high-performance file systems, plus access to standard UNIX file systems.

- Data parallel compilers, CM Fortran and C*, plus their supporting run-time system (CMRTS) and libraries. CMSSL, the Connection Machine Scientific Software Library, is an example of such a data par-

Copyright © 1995 Thinking Machines Corporation

allel library. Othes include visualization libraries such as CM/AVS and CMX11.

- CMMD, the CM-5's message-passing library.
- The Prism programming environment, a windows-based environment that facilitates the debugging and performance analysis of both data parallel and message-passing programs.

All this software is available to the programmer writing message-passing applications on the CM-5 (or CM-5E). The *CM-5 User's Guide* and the *CMMD User's Guide* explain how to use them in support of message-passing programming.

1.2 What Does CMMD Provide?

CMMD supports the message-passing programming model on the CM-5 and CM-5E systems, using functions designed to take best advantage of the CM architecture.

The CMMD library is callable from C, C++, C*, Fortran 77, and CM Fortran. Programs or program modules written in CM Fortran or C* utilize the vector units (VUs) on the CM-5. Other programs or modules use only the microprocessor on each node. (See Chapter 3 for discussion of the relevant architecture features of execution models.)

1.2.1 The CMMD Library

CMMD provides the following types of functions:

- message-passing functions
- informational functions and other auxiliary functions to support the point-to-point functions
- global operations
- local and global I/O operations
- active messages (similar to remote procedure calls)

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

- low-overhead block data transfers between nodes
- timing functions
- error-handling functions
- system control functions (for setting defaults, enabling and disabling interrupts, manipulating servers, etc.)

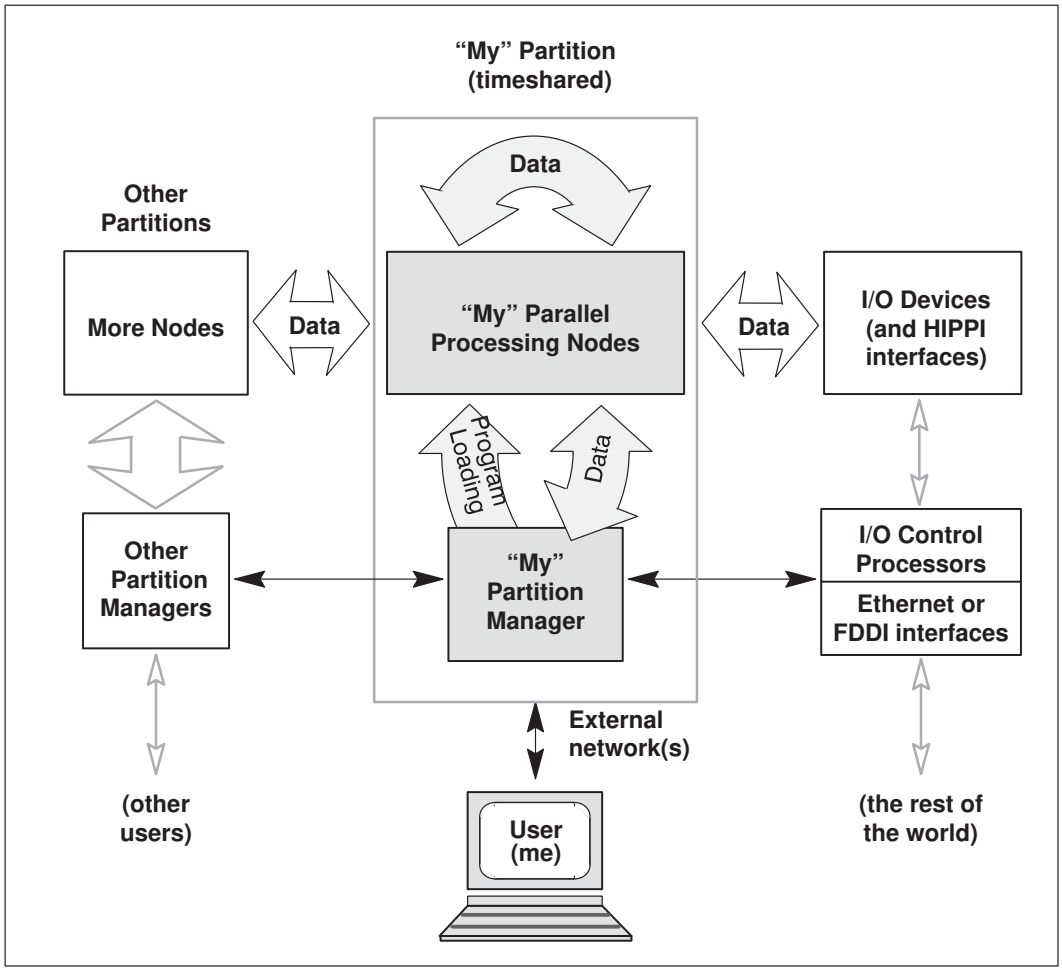


Figure 1. A user's view of a CM-5.

Users access a CM-5 system by running `rlogin` or `rsh` commands on a specific partition manager. A user program begins execution on the PM, downloads code to the nodes, then runs on nodes and PM both, passing data as needed among processors. Programs can exchange data with I/O devices or with other processes. In such cases the PM mediates the transfer.

Copyright © 1995 Thinking Machines Corporation

They are detailed in Chapters 4 through 20, and summarized in two quick reference guides, *CMMD for Fortran Quick Reference* and *CMMD for C Quick Reference.*

1.2.2 Communications Protocols

For general use, CMMD offers both synchronous and asynchronous communication utilizing a safe, convenient internal protocol. For advanced progammers, CMMD also makes available the underlying protocol-free messaging capabilities. For a fuller description of CMMD's message-passing facilities, see Chapter 2.

1.2.3 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 of the other nodes.
- 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. Only node 0 can write to the file.

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

CMMD I/O is discussed in detail in Chapter 15.

Copyright © 1995 Thinking Machines Corporation

1.3 A Sample CMMD Program

To create an application program using CMMD, the programmer writes a single program that is replicated and executed on all nodes of a CM-5 partition. CMOST treats this set of programs as a single process under the control of the partition manager (PM).

Here is a simple example of a CMMD "hello world" program, written in C:

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

main(argc,argv)
  int argc;
  char *argv[];
{
  CMMD_fset_io_mode(stdout, CMMD_sync_seq);
  printf("Hello world, from node %d\n",
         CMMD_self_address());
}
```

Once compiled, linked, and executed, this program prints the following on the user's terminal:

```
% hello_world
Hello world, from node 0
Hello world, from node 1
Hello world, from node 2 ...
```

The number of messages printed depends upon the number of nodes in the CM-5 partition. (The `CMMD_fset_io_mode` call places the standard output stream in synchronous sequential mode, so the messages appear in processor order, from 0 to the highest-numbered node in the partition.)

NOTE: The program above prints something from *every* node. You could also rewrite it to print a message from only a subset of nodes. For example:

```
CMMD_fset_io_mode(stdout, CMMD_independent);
fcntl(fileno(stdout), F_SETFL, O_APPEND);
if (CMMD_self_address() < 10)
  printf("Hello world, from node %d. \n",
           CMMD_self_address());
```


Copyright © 1995 Thinking Machines Corporation

Note the use of the `CMMD_independent` I/O mode. This is important, because `CMMD_sync_seq` mode requires that all nodes print something, while `CMMD_independent` allows the nodes to print as they wish.

1.4 Compatibility with Other Software

CMMD Version 3.3 requires CMOST Version 7.4 or higher.

CM Fortran Version 2.2 provides complete CM Fortran functionality with CMMD Version 3.3.

C* Version 7.2 provides complete C* functionality.

CM Fortran 2.1 and C* 7.1 provide partial functionality.

1.4.1 Header Files

C, C++, and C* programs using CMMD must include the header file `<cm/cmmd.h>`. Other standard header files, such as `<stdio.h>`, `<fcntl.h>`, and `<sys/types.h>`, may also be needed, depending on the particular program.

Fortran 77 and CM Fortran programs using CMMD must include the file `cmmd-fort.h`.

1.5 Using This Reference Manual

This *CMMD Reference Manual* provides information on the CMMD functions. Each chapter discusses some functional group of routines, or provides a more detailed explanation on the use or implementation of some type of routine. The chapters also provide reference writeups for each routine.

See the *CMMD User's Guide* for information on compiling, loading, and debugging message-passing programs.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation