Chapter 1

Introduction

CMMD is a library of message-passing routines for the Connection Machine CM-5 system. Programs that use CMMD typically operate in a message-passing style:

- Each processing node of the CM-5 runs an independent copy of a single program, and manages its own computations and data layout.
- Communication between nodes is handled by calls to CMMD message-passing functions.

CMMD allows you to send messages from one processing node to another in a number of different ways, depending on the needs of your application. CMMD provides functions for point-to-point messaging, global operations, active messages, and receive port operations (rports).

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, with the program compiled by the **`cmf`** compiler, and from C*, with the program compiled by the **`cs`** compiler. 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 2 for discussion of the relevant architecture features of execution models.)

1.1 Programming Models

CMMD Version 3.0 supports two programming models:

- The host/node programming model involves two simultaneously running programs. One program runs on the host, while an independent copy of a second program runs on each processing node. The host begins execution by performing needed initializations (including initializing the CMMD message-passing environment) and then invoking the node program; it may have much or little involvement in subsequent computations.
- The hostless programming model uses the host only to initiate and terminate execution and to act as an I/O server. A CMMD-supplied program performs these tasks; the user writes a single application, which runs on each of the nodes. The nodes communicate among themselves and synchronize themselves as needed.

At Version 3.0, hostless programs may be written in any of the supported languages. Host/node programs must have main programs written in a serial language, although these programs may call modules written in C*, CM Fortran, or DPEAC (the CM-5 VU assembler language).

For further information on these two programming models, see the *CMMD User's Guide.*

1.2 Communication Protocols

For general use, CMMD offers both synchronous and asynchronous communication that utilize a handshake protocol. For advanced progammers, CMMD also makes available the underlying protocol-free messaging capabilities.

1.2.1 Cooperative and Asynchronous Message Passing

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 pre-defined array into the channel; the receiving node reads the channel, then resets it for another use. No synchronization is needed.

Handshaking between Nodes

Each node typically has little knowledge of the data layout of any other node (except for functions and data structures that are common across all nodes). To establish a communications link between two nodes, therefore, a preliminary "handshake" step is required, in which the nodes exchange information about the type and size of data that is to be transferred, and the locations of the source and destination. This handshake step adds to the time taken by the transfer, but it does guarantee that messages go where they should, when they should.

1.2.2 CMAML: Active Messages

In some message-passing programs, the standard CMMD point-to-point and global communication functions are not sufficient. Some applications require an efficient communication protocol for a specific kind of data transfer not covered by the "standard" CMMD operators.

The CMAML (active message) layer of CMMD is a set of communication primitives on which the higher-level CMMD operations are based. These routines provide fast packet-based communication and simple low-latency array transfer.

With the active message facility, one node activates a 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.

With the related rport facility, one node defines a port for the reception of an array (indicating not only the characteristics of the array to be received, but whatever tasks the reception should involve), and a second node sends data to that port.

The CMAML layer is described in more detail in Chapter 11.

Usage Note: CMMD makes significant use of both active messages and low-level array transfer routines in its high-level message-passing functions. There is no need to program at the CMAML level to obtain the benefits of these routines; they are already implicit in the "standard" CMMD functions. The CMAML routines are intended for use by knowledgeable programmers who wish to implement custom communication protocols.

1.3 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.
- 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 applications to access very large files.

1.4 Supporting Utilities

Message-passing programs in the CM-5 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.

1.5 Message-Passing Applications

One of the most useful features of message-passing programs is that they allow the CM-5 processing nodes to synchronize as frequently or infrequently as required for a given application. Some applications, for example, require that all nodes work on particular procedures simultaneously, synchronizing regularly. Other applications may need the nodes to operate independently throughout most of the program, with little or no synchronization. Some applications need to share or update data synchronously across the nodes, while others can use asynchronous message passing, allowing the nodes to interrupt each other when they are ready to exchange data.

Message passing allows all these kinds of program styles and transaction protocols. Thus, a message-passing program is free to distribute program tasks and data among the nodes as necessary, using synchronization and data communication between nodes only when required

This arbitrary load balancing of tasks and data makes message passing particularly useful for applications that require dynamic allocation of tasks and data. Such applications typically use a class of algorithms known as node-expansion algorithms; for example, divide and conquer algorithms, branch and bound algorithms, asymmetric traveling salesman problems, and tree search problems.

1.6 More about Messages

Transmitting a message in a CMMD program involves one of four things:

- transfer of a block of data between nodes via a point-to-point message
- global broadcast, computation, or synchronization across all nodes

- transmission of a network packet that triggers a function call on the receiving node (that is, an *active message*)
- transfer of a block of data between nodes via an *rport* operation

In point-to-point message transmissions, the source and destination nodes of the message take equal roles in the exchange. The sending node, in its message-passing call,

- signals to its “partner” node its intention of sending the data
- specifies the node that is to receive the data
- provides a “user tag” to serve as a label for the data during the exchange
- specifies where in its memory the data is to be found

The receiving node, in its message-passing call,

- posts a receive buffer as being available to receive data
- specifies the node from which it expects to receive the message
- specifies the “user tag” it expects to see labeling the message

Once both nodes have signaled readiness, the data transfer takes place. The nodes may either wait for the operation to complete (a “blocking” or synchronized transmission), or they may simply perform their part of the operation and then resume other tasks, letting the transfer take place in its own time (in an asynchronous transmission).

Global message operations are similar, but require the participation of all the nodes at once. In a global operation, every node

- signals its readiness to participate in the global operation
- specifies where in its memory data is to be found and/or stored

Here, the operation takes place as soon as all nodes have signaled their readiness by making the appropriate function calls; it *cannot complete* for any one node until it is completed for *all* nodes. Thus, global messages of any type implicitly synchronize the nodes.

Active messages use a slightly different model of transmission:

- The source node sends a message via a function call that specifies most of the parameters of the message.
- On the destination node, the arrival of the message automatically triggers a function call.

Rports provide a simple interface for low-level, low-latency block memory moves. With these functions,

- The destination node allocates a port that defines the characteristics and handling of a block of data (an array) to be received, and informs the source node of the port.

- The source node calls a copy function that sends data to that port.

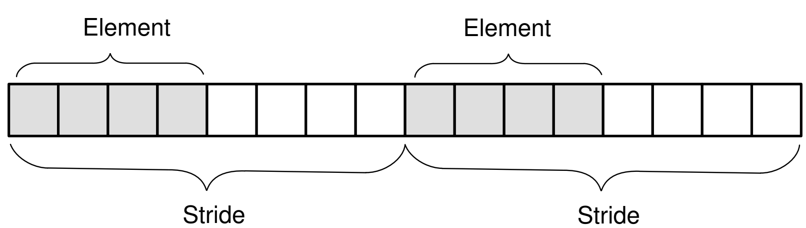
1.6.1 Array Types Used in Messages

The blocks of data transmitted by CMMD programs are typically in the form of arrays. These arrays come in three varieties:

- A *serial array* is a single block of data stored in a single node's memory.

  Serial arrays are also referred to as *block serial* arrays, to distinguish them from vector serial arrays.

- A *vector serial array* or *vector array* is an array that is spread out within a single node's memory, with a fixed amount of space between individual array elements. The distance from the start of one element to the start of the next is known as the *stride*.



Vector serial arrays allow a CMMD function to perform "gather/scatter" operations in tandem with communication, collecting or dispersing data across node memory without the need for temporary arrays.

Note: Vector array operators (indicated by a **_v** suffix) are provided as a convenience. These functions generally do not operate as efficiently as those for block serial and parallel arrays.

- A *parallel array* is an array of data stored across the memory of a node's VUs. (See Section 2.3.)

1.7 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 something like 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 each node. You could also rewrite it to only print a message from a subset of nodes. For example:

```
CMMD_fset_io_mode(stdout, CMMD_independent);
if (CMMD_self_address() < 10)
```

```
    printf("Hello world, from node %d. \n",
              CMMD_self_address());
```

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.

1.8 Compatibility with Other Software

CMMD Version 3.0 requires CMOST Version 7.2.

If CM Fortran is used with CMMD Version 3.0, it must be Version 2.1 or above; if C* is used, it must be Version 7.1.

IMPORTANT

Please note that the CMMD library is under continual development and hence subject to possibly substantial changes, which may affect its compatibility with previous versions and with other software packages.

1.8.1 Include Files

Programs using CMMD must include the include file **cm/cmmd.h**. Other standard include files, such as **stdio.h**, **fcntl.h**, and **sys/types.h**, may also be needed, depending on the particular program. CM Fortran programs should also include the file **cmmd_fort.h**.

1.9 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.

In addition, Appendix A provides a “Quick Reference” summary of CMMD routines for the C programmer. Appendix B provides an equivalent summary for the Fortran programmer.

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