Chapter 14

CMMD

The Connection Machine communication library, CMMD, is an advanced, user-extensible communication library.

- For portability, the CMMD library provides traditional message-passing functions that facilitate the porting of MIMD-style codes to the CM-5 and the design and creation, on the CM-5, of applications intended for execution on a variety of machines.
- For users designing applications specifically for the CM-5, it offers a wide variety of communication functions that supplement the CM-5's data parallel software, allowing the use of multiple programming models and techniques.
- For users with an experimental bent, it offers primitives with which they can design and construct new communication protocols.
- For all users, it offers great versatility. Based on active messages, CMMD provides synchronous and asynchronous functions, heavy-weight and light-weight functions, and global operations — allowing users to make the most effective use of CM communications for their applications' performance and efficiency.

Applications using CMMD take advantage of the level of programming control offered by node-level programming. This type of programming is particularly well suited to applications that demand dynamic allocation of tasks or data among processors.

Copyright © 1993 Thinking Machines Corporation

14.1 Node-Level Programming with CMMD

In node-level programming, a single program executes independently on each node; the nodes communicate only through explicit communication functions. CMMD provides for both synchronous and asynchronous communications functions; it thus provides interprocessor communication that falls outside the range of the data parallel languages.

CMMD permits concurrent processing in which synchronization occurs only

- between matched sending and receiving nodes, during the execution of cooperative communication functions
- among all nodes, during the execution of global communication or I/O functions
- when explicitly requested by a CMMD synchronization function

At all other times, computing on each node proceeds asynchronously.

CMMD can be called from applications written in Fortran 77, CM Fortran, C, C++, and C*. It handles both serial and parallel data.

In addition, CMMD offers both serial and parallel I/O facilities, based on UNIX I/O. In serial I/O, each node reads and writes files independently. (Opening and closing files may be done either independently or cooperatively.) In parallel I/O, the nodes cooperate to open, close, read, and write files: reading a file may send the same data to all nodes, or may distribute data across all nodes.

14.2 Programming Models

CMMD supports a variety of programming models.

14.2.1 Hostless and Host-Node Models

CMMD supports both the host/node programming model and the hostless programming model. In host/node programming, the user writes two programs: one runs on the host (a CM-5 partition manager), while independent copies of the node program run on each of the processing nodes. The host may have little involvement aside from initially invoking the node program and providing user

November 1993
Copyright © 1993 Thinking Machines Corporation

interface services. In the hostless programming model, the user writes a single program, independent copies of which run on every node. The host, meanwhile, acts as a server (usually an I/O server), executing a CMMD-supplied server program. (Users may customize this program, if they choose to do so.)

These two generic models allow the use of a wide variety of more specific programming models, from the highly asynchronous (such as many master-worker programs and programs using tree-based algorithms) to the highly synchronized.

14.2.2 The Global-Local Model

CMMD also supports the global-local programming model. In this model, the main program is written in CM Fortran and executes globally, laying out parallel arrays across all processing nodes. The global program calls user-written "local routines," written either in CM Fortran or in C plus DPEAC. The local routines perform node-level computations and use CMMD for inter-node communications. (In essence, they are hostless CMMD programs.) When they finish, they return control to the global program.

During the execution of local routines, each node operates on its own subgrid of a global parallel array as if that subgrid were a complete nodal array. Informational functions allow the node to locate the position its "subarray" within the global array. Local routines can also create arrays, common blocks, etc., of their own. These are completely local and are accessible only within the scope of the local routines.

The global-local model is thus useful for applications that can benefit from both global data parallel and local data parallel and control parallel techniques. It is one more example of the flexibility that marks Connection Machine programming.

14.3 Message-Passing Protocols

Message-passing protocols coordinate and control communications between two processors, operating either synchronously or asynchronously. Message passing can occur in cases in which each processor has no a priori knowledge of the data layout on other processors and cannot read or write remote memory location without "permission" from the remote node. It can also occur in more well-

Copyright © 1993 Thinking Machines Corporation

defined environments, where communication patterns are relatively static or where nodes have knowledge of remote memory layouts.

CMMD offers both functions that can handle the general case, where the processors have no knowledge, and functions that take advantage of well-defined situations. In all, the library offers four classes of functions:

- Point-to-point functions, which use an initial handshake (RTS/ACK) protocol to coordinate subsequent data transmissions.
- Functions that create and use long-lived virtual communication channels to support low-latency operations on messages of all sizes.
- Functions that support active messages and active message handlers; these provide an easily extensible mechanism for invoking functions on remote nodes via the transmission of an interrupting Data Network packet.
- Functions that use "receive ports" for low-latency transfer of arrays from one node to another.

14.3.1 Point-to-Point Functions

Point-to-point message-passing functions require coordination only between the sender and receiver. Both blocking and non-blocking functions are supplied. Blocking functions are "cooperative": they do not complete until the corresponding function has been called on the destination processor. (If the second processor is not ready, the first processor waits for it.) Thus, they provide synchronization as well as communication. Non-blocking functions are asynchronous: they return as soon as the processor has announced its readiness either to send or to receive. The processor can then perform other work, while waiting for the destination processor to announce its readiness. When both processors have signaled readiness, they receive interrupt messages telling them to begin transmission.

Point-to-point data transfers are performed in memory order. A transfer continues until either the entire source array is sent, or until the destination array is filled up.

On CM-5 systems equipped with vector units, CMMD point-to-point functions can transfer data from either microprocessor memory or vector unit memory. Consider this as the act of transmitting arrays of data from one processor to another. The functions operate on three types of arrays:

November 1993
Copyright © 1993 Thinking Machines Corporation

- *Serial arrays*. Stored in microprocessor memory, serial arrays are specified simply by a starting address plus a length. (These are very standard-appearing function calls, in which the programmer merely supplies the array's name plus a "buffer descriptor" argument that gives the array length, in bytes.)

- *Strided serial arrays*. Like serial arrays, these are stored in microprocessor memory. They are defined by a starting address, an element size, a stride, and an element-count (specifying the number of elements to be transferred).

  The stride is the distance (in bytes) between the starting point of one element and the starting point of the next element.

- *Parallel arrays.* Parallel arrays are CM Fortran or C* arrays, and are defined by the usual CMRT array descriptor data structures for such arrays. These reside in the memory of the vector units, usually spanning the memories of all four VUs. (The CMMD program does not specify this data structure; it merely supplies the array's name plus a "buffer descriptor" argument indicating that the array is parallel rather than serial.)

Serial transfers can mix serial arrays and strided serial arrays, to provide scatter-gather behavior. (See Figure 25.) Parallel transfers always transfer data between identically laid-out arrays. Programmers may use CM Fortran or C* functions to reshape arrays, or to change serial to parallel arrays, or vice versa, if they wish to send data between parallel and serial arrays, or between unlike parallel arrays.

Copyright © 1993 Thinking Machines Corporation

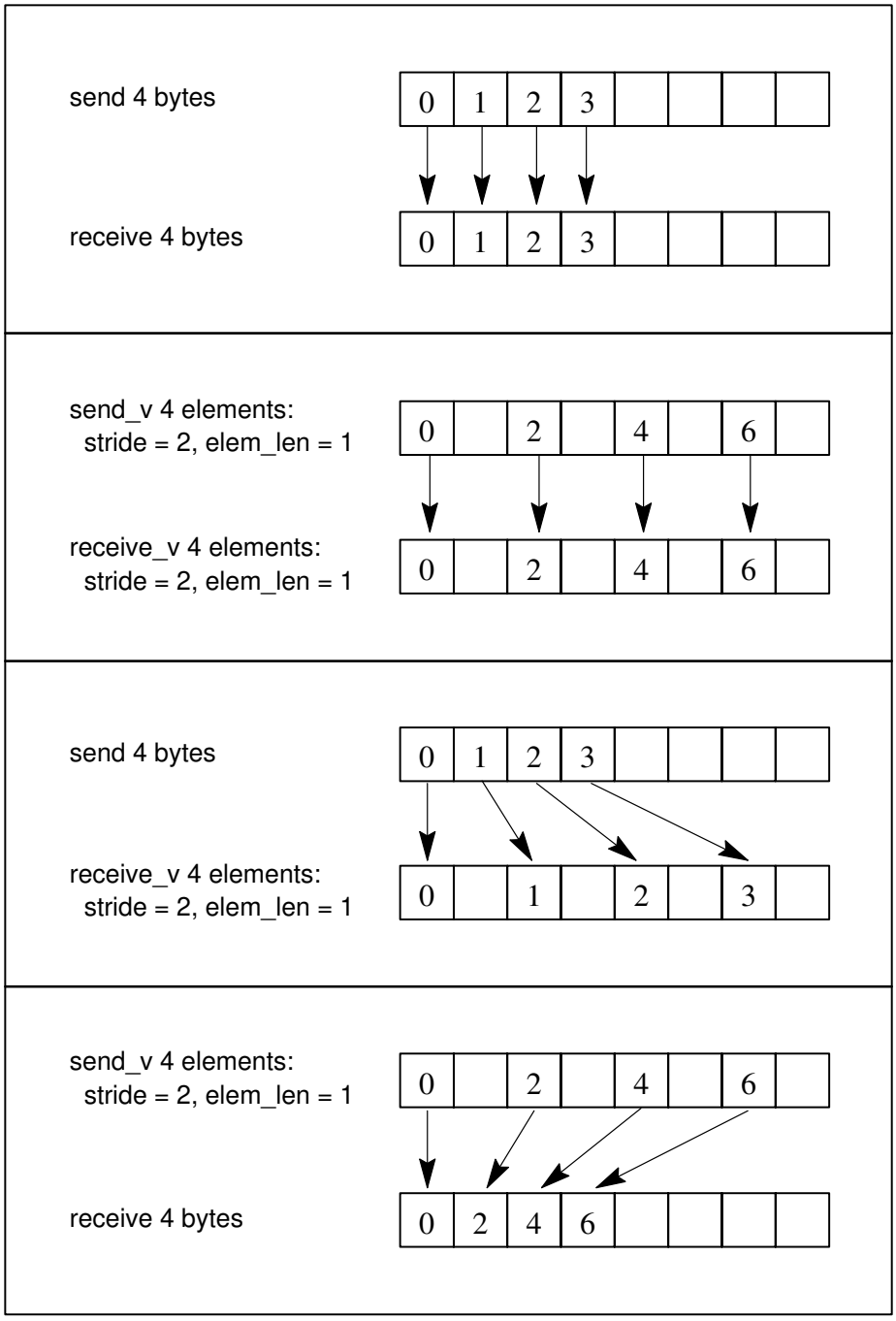

Figure 25. Sending and receiving data.

Global Operations

Copyright © 1993 Thinking Machines Corporation

The CMMD library also provides a number of global functions that operate under the same general protocols as the point-to-point functions. Global functions perform their operations over all the nodes, and require participation from all nodes in the call. Some include and some exclude the host. Some perform a single operation and return their result as a return value; others operate on vectors of data and write their results into destination buffers.

Global functions include

- broadcasting data or instructions from the host or from one node to all nodes
- reducing data from all nodes to all nodes or to the host
- performing scans (parallel prefix operations) across the nodes
- performing segmented parallel prefix operations
- concatenation of elements into a buffer on all nodes, or into a buffer on the host.

14.3.2 Virtual Channels

Communication patterns frequently remain constant over time. CMMD provides functions for opening, closing, and transmitting data via software communication "channels" for these cases. These channels are uni-directional connections between specific processors. When one is opened, the two processors that opened it exchange information about the relative array shapes at each end of the channel.

The channel remains active through as many uses as desired, allowing data to be transmitted without incurring the handshake overhead associated with traditional message-passing systems. By thus amortizing the overhead of establishing a channel over multiple uses, these functions allow programmers to efficiently operate on static communication patterns and to send both small and large amounts of data repeatedly between nodes with low latency.

Channels provide an implicit ordering or sequencing to transactions. Thus, they guarantee that messages are received in the order in which they are sent.

Copyright © 1993 Thinking Machines Corporation

14.3.3 Active Messages

Drawing on results of the TAM project at the University of California at Berkeley, active messages provide an easily extensible mechanism for invoking functions on remote nodes via the transmission of an interrupting Data Network packet.

The format of an active message consists of one word containing the address of a "handler function" to be invoked on the destination processor, followed by *n* words (with *n* defined in the function call itself) making up the argument list to be passed to that function.

Upon receipt of an active message, the destination processor invokes the specified "handler function," passing to it as arguments the contents of the message.

CMMD makes use of active messages to perform protocol-processing functions. In addition, it provides a set of primitives that allow users to create their own communication functions according to their own application requirements.

14.3.4 CMAML Array Transfers

In addition to active messages themselves, CMAML (the CMMD Active Message Layer) also supports the transfers of blocks of data (usually arrays) from node to node. Either serial or parallel data can be transferred from the source node into a receive port on the destination node. When the transport operation completes, the source and/or destination node may invoke an attached handler function.

Users familiar with the CMAML machinery will find it easy to construct a variety of additional communication and memory exchange functions, such as remote memory store and fetch, send-to-queue, store-with-op, and compare-and-swap.

As with most transport layers, CMAML functions assume that some higher layer of software is providing any needed protocol, and that (for example) receiving nodes know what to do with any data sent to them. Users must ensure that their applications provide such protocol.

November 1993
Copyright © 1993 Thinking Machines Corporation

14.4 I/O

CMMD provides the node-level programmer with I/O routines for opening, reading, and writing files using both standard UNIX I/O semantics and extensions that provide for parallel I/O. Support for UNIX functions is provided by linking an I/O server routine into the host program. A standard host program including the server is provided by the **cmmd-ld** link utility for users not wishing to write a host program.

Parallel extensions provided by CMMD apply to those UNIX I/O functions that take a file descriptor or stream as an argument, such as **read()**, **write()**, etc. These extensions coordinate the actions of nodes acting in concert so that the potential of CM-5 parallel I/O devices may be realized. UNIX calls that do not take a file descriptor, such as **mkdir()**, **chdir()**, **chmod()**, etc., do not synchronize and may be called by any node.

14.4.1 Support for Extra-Large Files

Since UNIX uses only a 32-bit integer file pointer, seek operations on very large files are limited. CMMD therefore supplements the standard UNIX operations with functions that take an argument of type double, indicating a position within an extra-large file.

14.4.2 I/O Modes

In order to support UNIX functionality while simultaneously extending it to support parallel operations, CMMD creates new file descriptors and introduces new I/O modes.

The I/O modes defined by CMMD are as follows (see Figure 26):

- local independent
- global independent
- global synchronous broadcast
- global synchronous sequential

Copyright © 1993 Thinking Machines Corporation

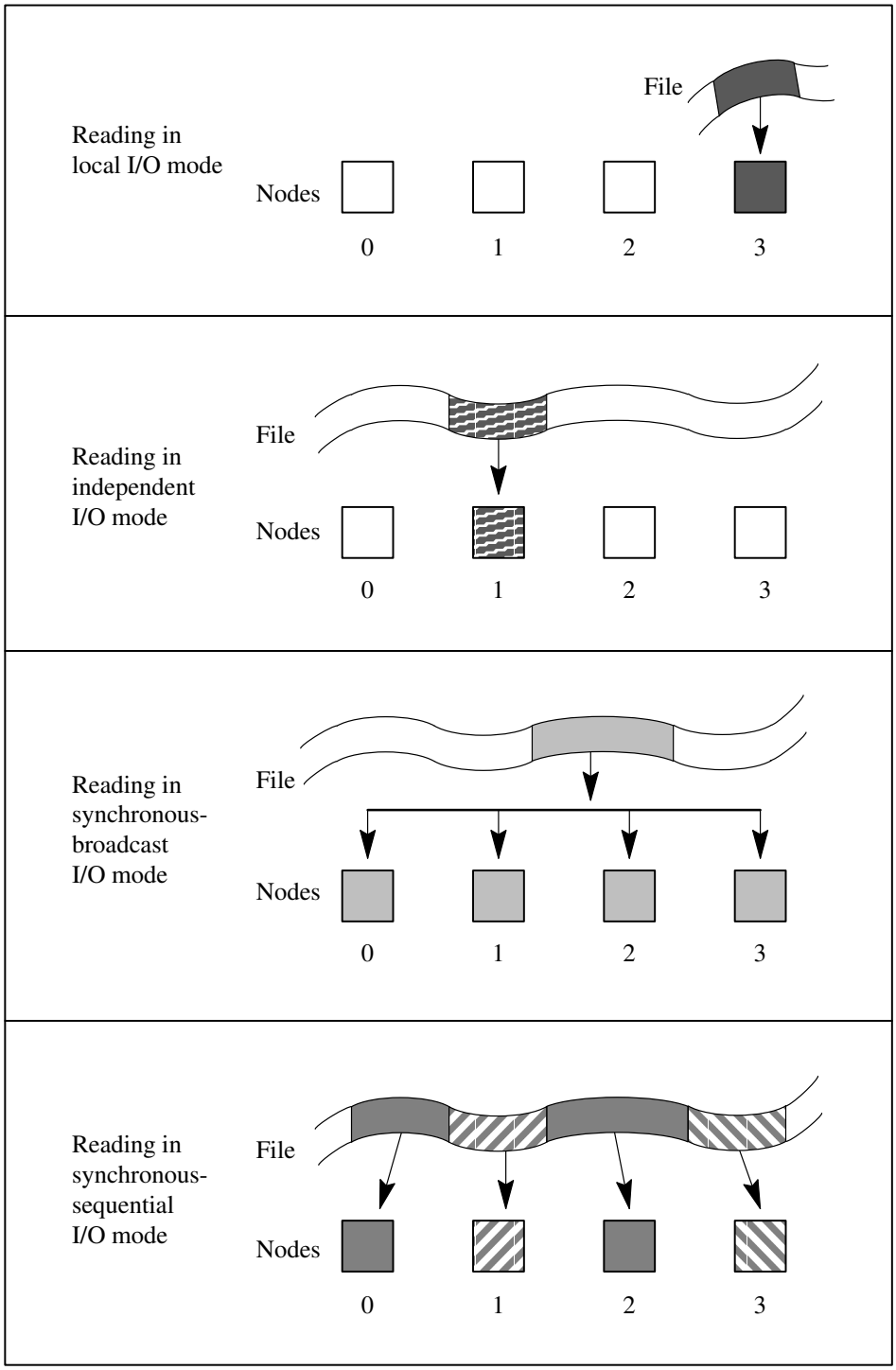

Figure 26. Four patterns for reading a file.

Copyright © 1993 Thinking Machines Corporation

Local Mode

Local mode supports completely independent I/O operations. In this mode, individual nodes can read and write different files independently, without having to coordinate activities with other nodes. Nodes maintain their own local file descriptors and file pointers.

Global Independent Mode

Global independent mode allows all nodes to access a single file for independent reading and writing. Files opened in this mode create only a single entry in the process's table of file descriptors. Every node, however, maintains its own pointer to the file, and can move that pointer about at will, reading or writing the file in an independent manner.

Local and global independent modes are used primarily to read ordinary files on UNIX file servers. The major reasons for choosing global independent mode over local mode are, first, the conservation of file descriptors, and second, the ability to change the mode of global independent files to one of the global synchronous modes, thereby achieving high-performance I/O.

Global Synchronous Broadcast Mode

Global synchronous broadcast mode allows nodes to simultaneously read the same data from a file or stream. Data, as it is read in, is broadcast to all the nodes. This is particularly useful for having all nodes read the same input from the user's terminal or read in the same header information from a file. On output, global broadcast mode acts as if only processor 0 contributed data. A file or stream in synchronous broadcast mode must be written or read by all nodes in the partition.

Global Synchronous Sequential Mode

Global synchronous sequential mode is similar to global synchronous broadcast mode, except that in the sequential case, data is distributed across the nodes instead of broadcast. Again, all nodes must participate in the I/O call.

For reads, each node issues a request for a buffer of a specified length. The data is read into the nodes as if node 0 first read, with all successive nodes following. For writes, each node contributes a buffer to be written, and the data is written as if node 0 first wrote its buffer, with each successive node's data immediately

Copyright © 1993 Thinking Machines Corporation

following. Note that the amount of data read or written may be different for each node. Nodes not having any data to contribute may set their buffer length to be zero.

Establishing and Changing Modes

Each file descriptor, when created, starts out in one of these modes. The mode can be changed by CMMD function calls as the program progresses. File descriptors for local files are created when an individual node opens a file; file descriptors for global files are created when all nodes open a file synchronously.

All UNIX functions that take a file descriptor or stream as an argument are sensitive to the associated I/O mode. The major difference concerns synchronization across nodes. In either of the independent modes, operations on file descriptors proceed independently. In either of the synchronous modes, operations on file descriptors synchronize across all nodes in the process of performing the operation.

November 1993
Copyright © 1993 Thinking Machines Corporation