Chapter 2

CMMD Messages

This chapter introduces the CMMD message-passing functions and sets out some guidelines for programming.

A CMMD application is a program written in a higher-level language (C-based or Fortran-based) that includes calls to CMMD library routines for message passing. Programs calling these routines have the following characteristics:

- Each processing node runs an independent copy of a single application program.
- Each node executes its copy of the application program at its own pace; it controls its data layout, manages its own memory and flow of control, and does its own computations.
- At intervals, as determined by the application program, the nodes communicate with each other: one or more nodes may send data to another node, or all nodes may join in a global operation such as a summation, a broadcast, or a synchronization.

2.1 Message Passing Functions

CMMD offers several types of message-passing functions: send and receive functions, virtual channel functions, active messages, and scopies and pcopies. This section introduces each in turn, and discusses briefly the mechanism employed by each.

Copyright © 1995 Thinking Machines Corporation

2.1.1 Send and Receive Functions

The simplest of the message-passing functions are the CMMD send and receive functions. Each of these functions sends a copy of some section of memory from one node to another. They are

```
CMMD_send_block[_v]
CMMD_receive_block[_v]

CMMD_send_and_receive[_v]
CMMD_swap[_v]

CMMD_send_async
CMMD_receive_async

CMMD_send_noblock
```

The *blocking functions* (those with `_block` in their name) automatically synchronize the sending and receiving nodes at the time of communication. Whenever a blocking function is called, the calling node waits until the message transmission is complete. (If the intended partner node does not make a complementary call, i.e., a matching receive for a send, or a matching send for a receive, then the first node waits forever.) While the calling function is waiting, it polls the Data Network continuously for incoming messages, but does nothing else during this time. Thus, from the calling node's point of view, message transmission always occurs at the exact line in the program where the blocking function is invoked.

The *asynchronous functions* (those with `_async` or `_noblock` in their name) do not synchronize the nodes. Rather, the function simply "posts" its intention to send or receive, then returns to the user program. Execution continues until a partner node makes a complementary call, using either a blocking function *or* an asynchronous function (there is no restriction against mixing the two types in a single transaction). The actual data transmission then takes place while the user code continues to run. Depending on whether the user has disabled CMMD interrupts (see Chapter 11), this transmission may occur during interrupts or during explicit calls to CMMD polling functions.

A program that calls an asynchronous function may do useful work until the partner responds. This contrasts with calling a blocking function, which prevents the program from proceeding until the entire message has been transferred. When a program does useful work while waiting for an asynchronous message to complete, it is often said to "hide latency" or to "overlap

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

communication with computation." CMMD provides information functions that allow the user to issue queries as to the completeness of any asynchronous transfer.

2.1.2 Virtual Channels

Virtual channels improve the performance of repeated patterns of message passing, by amortizing overhead over multiple transactions.

The blocking and asynchronous functions described above automatically exchange preliminary information (a "handshake") between the sender and receiver before the message itself is transferred. The handshake is invisible to the user, except for the latency it incurs. When the same sender-receiver pair communicates a series of messages, it is often advantageous to use a *virtual channel*. A virtual channel performs just one handshake, at the outset, rather than one before each message, thus amortizing the latency. (For details about the handshake protocol, see Section B.2 in Appendix B.)

Once a virtual channel has been *opened* at both ends, the sending node *writes* a predefined array in to it. The receiving node *reads* the channel, then *resets* it, thus signaling that it is ready to receive the next message. Any number of messages can be sent through the channel, one per reset. When the nodes are finished using the channel, they *close* their respective ends.

2.1.3 Active Messages

Active messages work by invoking an application-defined *handler function* on the receiving node. The handler function executes as soon as the active message is received. (See Figure 3.) Active messages do not perform a handshake (see Section 2.1.2), and are thus often termed *protocol-less*. In fact, one of their most common uses is for *implementin*g protocols such as the handshake mentioned above.

A *short active message* (available on both the CM-5 and the CM-5E) transfers four words of data, which the handler is free to use as it wishes. A *long active message* (available only on the CM-5E) can transfer up to 17 words of data; thus it can be viewed as protocol-less "mini-block transfer."

Copyright © 1995 Thinking Machines Corporation

2.1.4 Protocol-less Block Transfers: scopies and pcopies

CMMD also provides *scopies* (pronounced "ess-copies") and *pcopies* (pronounced "pea-copies"), which transfer blocks of data with even less overhead than virtual channels. The blocks can be of arbitrary size, in contrast with long active messages (see Section 2.1.3). In an scopy or pcopy, the receiver sets up an *rport*, which points to a segment of memory, and which is used for keeping track of the transaction. The receiver then either enables interrupts or polls explicitly for incoming data. The sender (or senders, as there may be an arbitrary number of them) calls an scopy or pcopy function with the rport as one of the arguments. The data is transferred directly from the sender's buffer to the buffer pointed to by the rport.

CMMD performs no handshake for scopies or pcopies. Instead, the user is responsible for arranging any necessary coordination or synchronization (perhaps by exchanging active messages, or perhaps by simply building any needed information into the code).

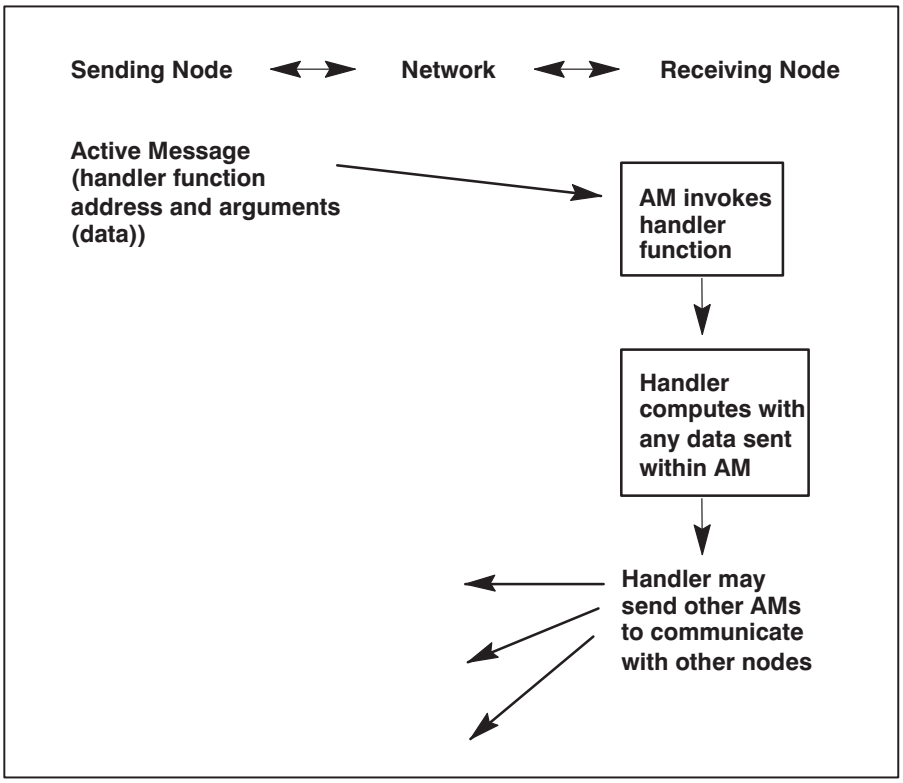

Figure 2. Active messages: protocol-free communication.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

All active messages, scopies, and pcopies are part of the *CMAML layer* (pronounced "sea mammal"), all functions of which begin with the `CMAML` prefix. CMAML is intended for use by knowledgeable programmers who wish to create their own custom functions, protocols, and so on. *The CMAML interface is considered* volatile, *and is subject to change at any time.*

2.2 Programming Guidelines

Here are some important general rules about programming with CMMD message passing:

Rule #1: For asynchronous messages, receive first, send second.

Receiving data tends to reduce the load on the Data Network. Sending data tends to increase it. Therefore, when you attempt to receive first, you unload the Data Network, so that subsequent sends are likely to perform better.

Rule #2: Keep receive calls and send calls balanced.

The number of pending (i.e., "already posted") asynchronous sends or receives on a given node is limited to about 127 (see Chapter 13 for details). Allowing pending sends or receives to build up causes the asynchronous functions to eventually return an error code and do nothing. In fact, the shortest possible *failing* CMMD program might look something like this:

Node A does this:

```
FOR I=1 to 300
CALL CMMD_SEND_ASYNC(NODE_B, <other args>)
END

FOR I=1 to 300
CALL CMMD_RECEIVE_ASYNC(NODE_B, <other args>)
END
```


Copyright © 1995 Thinking Machines Corporation

While node B does this:

```
FOR I=1 to 300
CALL CMMD_SEND_ASYNC(NODE_A, <other args>)
END

FOR I=1 to 300
CALL CMMD_RECEIVE_ASYNC(NODE_A, <other args>)
END
```

The reason it would fail is that the asynchronous functions would return (undetected!) error values after the first 127 or so, and would thereafter do nothing.

Your best strategy for asynchronous message, therefore, is to combine rules 1 and 2:

Keep send and receive calls balanced, make receive calls first.

Here is a code fragment that illustrates this discipline. It accomplishes the same objective as the last code fragment, but it does so efficiently and correctly:

Node A does this:

```
FOR I=1 to 30

  FOR I=1 to 10
    ERR = CMMD_RECEIVE_ASYNC(NODE_B, <other args>)
    IF( ERR .EQ. -1 ) THEN
          <take appropriate action>
    END
  END

  <optionally perform useful computation here>

  FOR I=1 to 10
    ERR = CMMD_SEND_BLOCK(NODE_B, <other args>)
    IF( ERR .EQ. -1 ) THEN
       <take appropriate action>
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

```
    END
  END

END
```

Node B does this:

```
FOR I=1 to 30

  FOR I=1 to 10
    ERR = CMMD_RECEIVE_ASYNC(NODE_A, <other args>)
    IF( ERR .EQ. -1 ) THEN
          <take appropriate action>
    END
  END

     <optionally perform useful computation here>

  FOR I=1 to 10
    ERR = CMMD_SEND_BLOCK(NODE_A, <other args>)
    IF( ERR .EQ. -1 ) THEN
     <take appropriate action>
    END
  END

END
```

This code fragment

- puts receives before sends
- keeps sends and receive *balanced*, i.e., it alternates between doing a few receives, then a few sends, etc.

  NOTICE: The example also checks the CMMD message passing function return values. *This practice is strongly recommended*.

Rule #4: Do your initial coding with CMMD send and receive functions or virtual channel functions.

Use the send and receive or virtual channel functions (sometimes called the “high level” or “heavyweight” functions) when starting out. By doing so you

Copyright © 1995 Thinking Machines Corporation

automatically take advantage of CMMD's built-in "handshake," plus many other safety features. Active messages, scopies, and pcopies — although they offer better performance under some circumstances — are much more difficult to use and debug. It is recommended that you attempt to use them only after gaining a good amount of experience with the higher-level functions.

Rule #5: When you do use active messages, keep your handler functions short and efficient.

Keeping handler functions short and efficient minimizes the amount of time that that handler suspends a node's normal data-receiving (and therefore Data Network unloading) activities.

Rule #6: Obey *all* rules for handlers and callbacks (detailed in Chapter 14).

Handler functions and callback functions may in general be invoked within the dynamic scope of the CMMD interrupt handler. For this reason, they must take care not to interfere with the state of the interrupted program (trashing registers, etc.). The programmer is fully responsible for avoiding such interference. A detailed set of rules for writing correct handlers appears in Chapter 14, and these should be carefully heeded.

Rule #6: Synchronous functions have their benefits.

When you are doing lots of sends and receives, especially well-patterned ones, blocking functions may actually outperform asynchronous ones. Whether or not this is true depends on the application, but you can reason about the question by considering an analogy between messages traveling through the Data Network and many parades marching simultaneously through the streets of a city: if the marchers are well coordinated, they can avoid jam-ups (such as would happen if they were all to converge on a single intersection — a single "node"!) at once. Such coordination, in the world of message passing, is generally easier to bring about with blocking functions than with nonblocking functions.

Generally speaking, debugging is easier with blocking functions, since the point at which data transmission occurs is always the point in the program where the blocking call is made.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Rule #7: Asynchronous functions have their benefits.

As was touched upon earlier, asynchronous functions allow the user to *overlap computation with communication*, i.e., to do useful work while waiting for the partner node to comply. Sometimes it is possible to write a program using all blocking functions, which are easier to debug, and then gain significant performance improvements by changing some calls to their asynchronous versions.

Finally, some communication patterns can simply be written using only blocking functions. The canonical example is a ring shift:

```
IF( SELF .EQ. LOWEST_NODE ) THEN
  ERR = CMMD_RECEIVE_BLOCK(HIGHEST_NODE, <other args>)
  <check return value>
  ERR = CMMD_SEND_BLOCK(SELF+1, <other args>)
  <check return value>
ELSIF ( ME .EQ. HIGHEST_NODE ) THEN
  ERR = CMMD_RECEIVE_BLOCK(SELF-1, <other args>)
  <check return value>
  ERR = CMMD_SEND_BLOCK(LOWEST_NODE, <other args>)
ELSE
  ERR = CMMD_RECEIVE_BLOCK(SELF-1, <other args>)
  <check return value>
  ERR = CMMD_SEND_BLOCK(SELF+1, <other args>)
  <check return value>
END
```

Clearly, such a program will "hang" immediately: none of the receives can proceed until their partner sends to them, yet — since the receives are *blocking* — nobody can send until their partner receives!

The solution is to use asynchronous receives:

```
IF( SELF .EQ. LOWEST_NODE ) THEN
     ERR = CMMD_RECEIVE_ASYNC(HIGHEST_NODE, <other args>)
     <check return value>
     ERR = CMMD_SEND_BLOCK(SELF+1, <other args>)
     <check return value>
ELSIF ( ME .EQ. HIGHEST_NODE ) THEN
```


Copyright © 1995 Thinking Machines Corporation

```
        ERR = CMMD_RECEIVE_ASYNC(SELF-1, <other args>)
        <check return value>
        ERR = CMMD_SEND_BLOCK(LOWEST_NODE, <other args>)
    ELSE
        ERR = CMMD_RECEIVE_ASYNC(SELF-1, <other args>)
        <check return value>
        ERR = CMMD_SEND_BLOCK(SELF+1, <other args>)
        <check return value>
    END
```

This program runs to completion: the sends can proceed, because the receives will "get out of the way" immediately. Similarly, the receives will eventually finish, because their partner sends are able to start.

NOTE: This example could also be written with a *single call* to `CMMD_send_and_receive`. The principle that it illustrates is valid nevertheless.

Copyright © 1995 Thinking Machines Corporation