Chapter 3

Programming Models and Execution Models

3.1 CM-5 Architecture

A CM-5 system consists of a large number of processing nodes, a small number of partition managers (PMs), some number (16−2K) of I/O control processors (IOCPs) and I/O devices. These components are all linked together by two internal communications networks, the Data Network and the Control Network. (See Figure 3.)

The Data Network is used for fast, low-latency, high-bandwidth communication of data between the processing nodes, PMs, and I/O devices. The Control Network provides a number of global control operations, such as broadcasts, scan/reduction operations, and node synchronization.

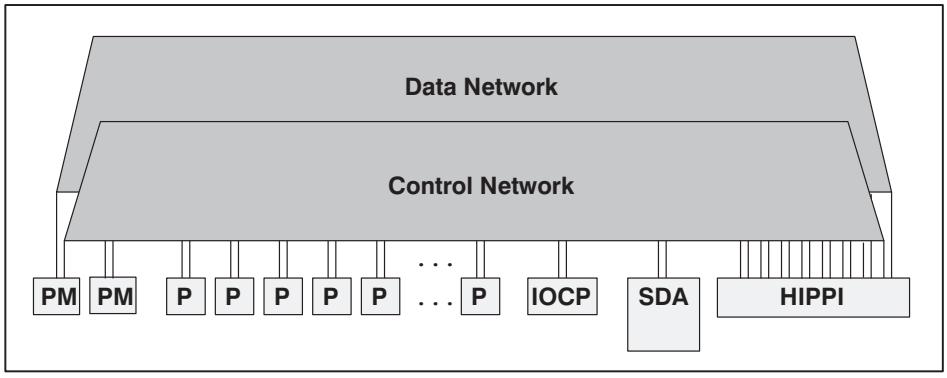


Figure 3. Components of a typical CM-5 system.

Each processing node contains a RISC microprocessor, 32 or 128 Mbytes of memory, and a Network Interface (NI) chip that connects it to the networks.

Copyright © 1995 Thinking Machines Corporation

Each processing node may also contain four vector processor units (VUs), which are located between the RISC microprocessor and the node's memory. The VUs, if present, provide highly efficient memory-based arithmetic calculations. (See Figure 4.)

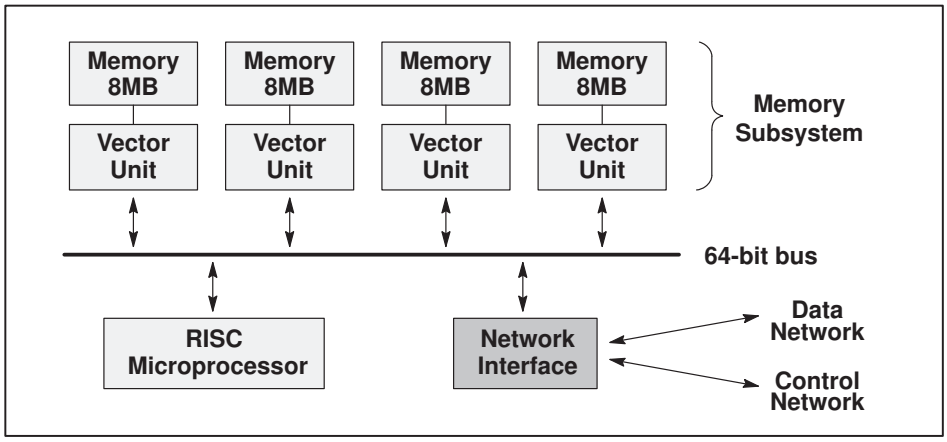


Figure 4. Components of a CM processing node.

3.2 Programming Models

The CM-5 architecture supports both data parallel and task parallel programming, as described in Chapter 1. This section describes these models in more detail, and discusses how CMMD relates to them.

3.2.1 Global Data Parallel Programs

Data parallel programs are often said to take a *global* view of the CM-5 system. The nodes operate mostly in lock-step, largely under the guidance of the partition manager. The partition manager executes serial code and stores serial data.

For global programs, the partition manager also executes serial code and stores serial data.

Data parallel programming on the CM is facilitated by the CM's data parallel languages, CM Fortran and C*. These languages may be regarded as parallel extensions of standard serial programming languages. Each pro-

Copyright © 1995 Thinking Machines Corporation

vides some method for defining *parallel variables*, in which the data of a single object is automatically stored across all the nodes. CM Fortran, for example, uses the Fortran 90 array features. Once the parallel variables (arrays, matrices, structures, etc.) are defined, a single sequence of instructions causes operations to be performed concurrently either on the full data sets or on selected sections of them. Thus they are often described as having a single *thread of control.*

3.2.2 Local Programs

Message-passing programs use a *local* view of the system. Each program still uses the whole CM-5 partition, with the PM initiating execution and providing I/O support. Each node, however, runs its own copy of the program, independently of the other nodes. The program thus has multiple threads of control, one for each node.

A program may perform *internode communication* using a wide variety of CMMD functions, which may *synchronize* some or all nodes at various points in the program's execution. Synchronization among nodes may occur either when the application invokes a CMMD *global function* (see Chapter 9), or whenever nodes communicate with each other using blocking message-passing functions.

If synchronization is improperly written, it can create *deadlocks*, in which two or more nodes are stuck forever (because each is blocking until some of the others proceed; see Chapter 2).

One of the most useful aspects of message-passing programming is that it is possible for the nodes to synchronize as frequently or infrequently as required, depending on the needs of the 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.

Scalability, provided by the CMRTS for data parallel programs, is possible (and desirable) in message-passing programs, but is not as automatic as it is in data parallel languages that distribute virtual processors' data among the

Copyright © 1995 Thinking Machines Corporation

nodes. CMMD provides functions that allow message-passing programs to be written in scalable fashion, discovering the size partition on which they are running and handling data layout and task distribution accordingly. Taking advantage of these features, however, is the programmer's responsibility.

3.2.3 Global/Local Programs

Global/local programs combine the benefits and techniques of data parallel and message-passing programming. Such programs have a main program written in CM Fortran or C*. This program arranges parallel data among the nodes (mostly invisibly to the user), and acts upon them in global fashion.

When local (on-a-node) control is required, the global program calls one or more *local subroutines*. These subroutines execute in *local mode*. Each treats its node's portion of a parallel variable as if it were an ordinary variable, operating on it in a single node's memory as needed. Each can also create its own local data structures (local arrays, for example) and operate on them. At the end of the local subroutines, control returns to the global program.

Global/local programming requires a detailed understanding of how data parallel languages arrange parallel data in the nodes' memory, and of how to compute their addresses, etc. This information is normally invisible to the data parallel programmer. In fact, one of the primary motivations behind data parallel programming is to liberate the programmer from such concerns. Nevertheless, in some situations it may be very useful to take advantage of the "global knowledge" and to "drop down" into local mode. If the programmer wants to do this, CMMD is there to allow inter-node communication to still take place.

3.3 Partitions and Addresses

The CM-5 processing nodes are grouped into one or more partitions, each controlled by a partition manager. Each application program runs on a single partition, with the partition manager serving as the host processor.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Each node within a partition has a *logical address* (often called a *node identifier*, a *node address*, or simply an *address*) relative to that partition, from 0 to (partition-size − 1). The host (PM) also has a logical address, always higher than any node addresses.

The nodes (and the host) use these logical addresses when sending messages to each other. They use three functions to obtain information relevant to logical addresses: `CMMD_self_address`, `CMMD_partition_size`, and `CMMD_host_node`. See Chapter 5 for information on these functions.

3.4 The CM-5 Execution Model

An application program executing on a CM-5 actually consists of two separate programs. One program runs on the host (the partition manager); the other runs on all the nodes in the partition.

3.4.1 Hostless and Host/Node Programs

CMMD supports two message-passing models:

- The *host/node* programming model involves two user-written programs. One program runs on the host, while an independent copy of a second program runs on each 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 program, which runs only on the nodes. The nodes communicate among themselves and synchronize themselves as needed. (**NOTE:** Local subroutines within global/local programs must be written using the hostless model.)

Hostless programs may be written in any of the supported languages. Host/node programs (described in detail in Chapter 19 and in the *CMMD User's Guide*) must have host parts written in a serial language, although these programs may call modules written in C*, CM Fortran, or DPEAC.

Copyright © 1995 Thinking Machines Corporation

3.4.2 Memory Maps

Both host and node programs look like standard UNIX executables. Each has its text region, heap, multiple stacks, and so on. The starting address and the extent of each of these regions in the node program is maintained uniformly across all nodes.

The address of any function is therefore identical on all nodes (but *not* on the host). This allows one node to trigger execution of a function on a second node, as in the case of active messages and their handler functions.

3.5 Data Storage

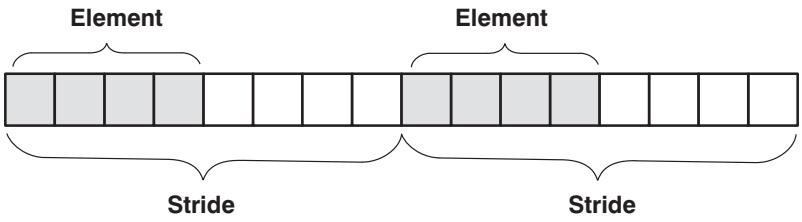
When a program executes on the CM, it stores its data in the memory of the processing nodes. In CMMD terminology, variables or other blocks or data are usually called *arrays.* Three types of arrays exist:

- 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.

  (Remember: In global data parallel programs, serial arrays are stored on the partition manager, which functions as a host processor for these programs.)

- 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

Copyright © 1995 Thinking Machines Corporation

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 3.6.)

3.6 Languages

CMMD programs can be written in standard Fortran 77, C, or C++. Programs written in these languages execute exclusively on the microprocessor in each node, ignoring the vector units. They exchange data exclusively in the form of serial and vector serial arrays. (See Figure 5.)

CMMD programs can also be written in the CM's data parallel programming languages, CM Fortran and C*. Programs written in parallel languages use both the microprocessor *and* the vector units; they store serial arrays in microprocessor memory and parallel arrays in VU memory. (See Figure 6.)

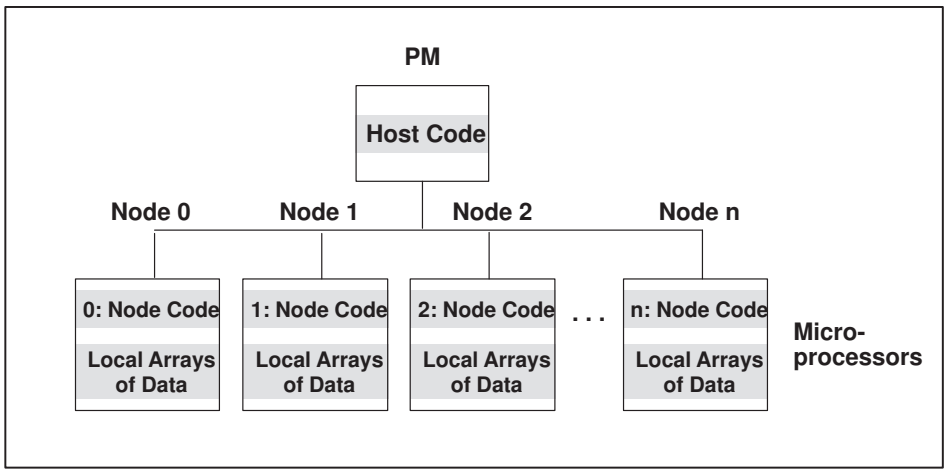


Figure 5. Execution model without VUs.

Copyright © 1995 Thinking Machines Corporation

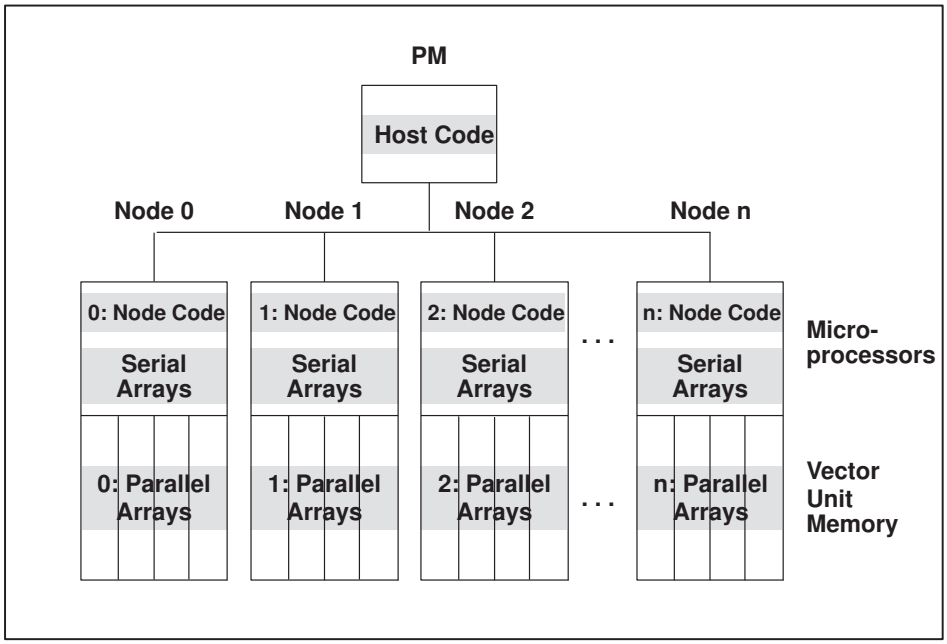


Figure 6. Execution model with VUs.

Similarly, programs written in C, C++, or Fortran 77 can call modules written in DPEAC assembly language (or in C/DPEAC), in order to access VU memory.

A parallel array in a "node-level" CMMD program is stored on a single node, with its values spread across the memory of the node's four vector units. It can thus take advantage of the VUs for high-speed floating-point computations. These parallel arrays can be passed as arguments to CMMD functions in much the same way as serial arrays.

3.6.1 Array Storage in Global, Local, and Global/Local Programs

Users of CM Fortran and C* should note the distinction between a purely global program (in the traditional style), a purely nodal program (with CMMD), and a global/local program. Let us take a CM Fortran program as an example. (A C* program would be essentially the same, except for some differences in terminology.)

Copyright © 1995 Thinking Machines Corporation

In a global CM Fortran program,

- serial arrays (FE arrays) are stored on the PM
- parallel arrays (CM arrays) are global, laid out across all VUs in the partition.

In a nodal CM Fortran program,

- serial arrays (FE arrays) are stored on each node
- parallel arrays (CM arrays) are local and are laid out across the node's four vector units.

For global/local programs, both models apply. The global program lays out the parallel arrays across all the VUs in the partition, and stores serial data on the PM. When a local function is invoked, each node treats its subgrid of the global array as a complete array, and operates on it however it wishes.

The global program stores serial data on the PM. This data is available to local functions only if the global program has first copied it into node memory. The local function can alter the node's copy of the data, but cannot affect the original data, which remains on the PM.

Certain restrictions apply to CM Fortran programs that use CMMD. These restrictions are detailed in Chapter 15 and in the *CMMD User's Guide.*

3.6.2 C*

Like "CM Fortran on a node," "C* on a node" distributes each array across the four VUs of a single node; the arrays are local, not global, in nature.

Unlike CM Fortran, C* requires a slightly different syntax for sending or receiving a parallel array than for sending or receiving a serial array. This syntax is explained in the discussions of the relevant functions later in this manual. C* uses the variables' declarations to distinguish serial from parallel arrays.

Programs written in C* should use the hostless model. The main node programs must be written in C*.

Copyright © 1995 Thinking Machines Corporation

3.6.3 DPEAC

C and Fortran code written for CM-5 systems can include assembly-language routines. The assembly language for CM-5 systems with VUs is DPEAC. DPEAC instructions allow a node's VUs to operate in parallel on separate data. All four VUs (or some subset) perform the same operation at the same address, each in their individual memory bank.

An application program module coded in DPEAC uses system calls to allocate parallel memory, then uses DPEAC instructions to manipulate data within that memory. The address returned by the allocation instruction may be passed as an array address to any CMMD message-passing function (but not to the global functions). The syntax is the same as if a block serial function were being transmitted; the CMMD system software determines from the array's address whether the array is indeed a block serial array or whether it is an array in VU memory.

DPEAC instructions, and their use, are documented in the *DPEAC Reference Manual*. The use of DPEAC from CMMD is discussed in the *CMMD User's Guide.*

Copyright © 1995 Thinking Machines Corporation