THINKING MACHINES CORPORATION

CONNECTION MACHINE TECHNICAL SUMMARY

The
Connection Machine
System

Connection Machine Model CM-2 Technical Summary

Version 6.0
November 1990

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, November 1990

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/ 876–1111

Contents

Part III Parallel Architecture

Connection Machine Model CM-2 and DataVault System

The Connection Machine Model CM-2 uses thousands of processors operating in parallel to achieve peak processing speeds of above 10 gigaflops. The DataVault mass storage system stores up to 60 gigabytes of data.

Part I
The Parallel Environment

Chapter 1

Parallel Architecture

The Connection Machine Model CM-2 is a data parallel computing system. Data parallel computing associates one processor with each data element. This computing style exploits the natural computational parallelism inherent in many data-intensive problems. It can significantly decrease the execution time of a problem, as well as simplify its programming. Execution time is frequently reduced in proportion to the number of data elements in the computation; programming effort is reduced in proportion to the complexity involved in expressing a naturally parallel problem statement in a serial manner.

The Connection Machine Model CM-2 is an integrated system of hardware and software. The hardware elements of the system include front-end computers that provide the development and execution environments for the users' software, a parallel processing unit of 64K processors that executes the data parallel operations, and a high-performance data parallel I/O system. Software elements begin with the standard operating system and program development environment of the front-end computer and enhance that environment with extensions to standard languages and tools that facilitate data parallel program development. Users write programs using familiar languages and constructs, taking advantage of the full, enhanced front-end development environment. When they choose, they can also call on CM language features and library routines specifically designed to handle tasks and problems germane to large-scale data-intensive programming. Programs have normal sequential control flow; new synchronization structures are not needed. Thus, users can easily develop programs that exploit the power of the Connection Machine hardware.

1.1 System Organization

The Connection Machine system was specifically designed to handle the largest computational problems. At the heart of any large computational problem is its data set: some combination of related data objects, such as numbers, characters, records, structures, and

arrays. The task of any application is to select, combine, rearrange, and operate upon this data. Data-level parallelism expedites this task by taking advantage of the parallelism inherent in large data sets.

At the heart of the Connection Machine system is the parallel processing unit, which consists of up to 64K processors, each with up to 128 kilobytes of memory. These processors can not only process the data stored in their memory, but also can exchange information among themselves and with I/O peripherals. All these operations happen in parallel on all processors.

A CM-2 parallel processing unit may contain 16K, 32K, or 64K data processors. The model CM-2a may contain 4K or 8K data processors. Here, and throughout this document, "K" stands for 1024, or 2^{10}. Thus 64K means 65,536; 32K means 32,768; 16K means 16,384; 8K means 8,192; and so on.

The Connection Machine processors are used whenever an operation can be performed simultaneously on many data objects. Data objects remain in the Connection Machine memory during execution of the program and are operated upon in parallel. This model differs from the serial model, where data objects in a computer's memory are processed one at a time, by reading each one in turn, operating on it, and then storing the result back in memory before processing the next object.

Of course, some small part of an application's data may be better processed serially. Such data resides in the memory of the front-end computer and is processed serially in the usual way. The memories of the parallel processors hold the bulk of data that can be usefully processed in parallel. The flow of control is handled entirely by the front end, including storage and execution of the program and all interaction with the user and/or programmer. Parallel data is operated upon through commands sent by the front end to the Connection Machine processors.

1.2 Data Parallel Hardware

The Connection Machine system implements data parallel programming constructs directly in hardware and microcode. Parallel data structures are spread across the data processors, with a single element stored in each processor's memory. When parallel data structures contain more data elements than the system has processors (the normal situation), the system operates in virtual processor mode, presenting the user with a larger number of processors, each with a correspondingly smaller memory. This allows the user to write programs assuming the number of processors that is natural for the application, rather than forcing code to conform to the number of hardware processors available. Each

hardware processor is made to simulate the appropriate number of virtual processors; as the program issues each parallel instruction, microcode causes it to be executed many times, once for each virtual processor. The same program can run without change on different quantities of hardware processors—but the more hardware, the faster it runs.

Interprocessor communication is implemented by a special-purpose high-speed network. When data is needed, it is passed over the network to the appropriate processors. Processors that hold interrelated data elements store pointers to one another, thus supporting completely general patterns of communication. In addition, special hardware supports certain commonly used regular patterns of communication. Nearest-neighbor communication in a multidimensional rectangular grid is particularly efficient.

High-speed transfers between peripheral devices and Connection Machine memory take place through the Connection Machine I/O system. All processors, in parallel, pass data to and from I/O buffers. The data is then moved between the buffers and the peripheral devices. Connection Machine high-speed peripherals include the DataVault mass storage system, the Connection Machine graphics display system, the Connection Machine VME I/O system, and the HIPPI high-speed interface.

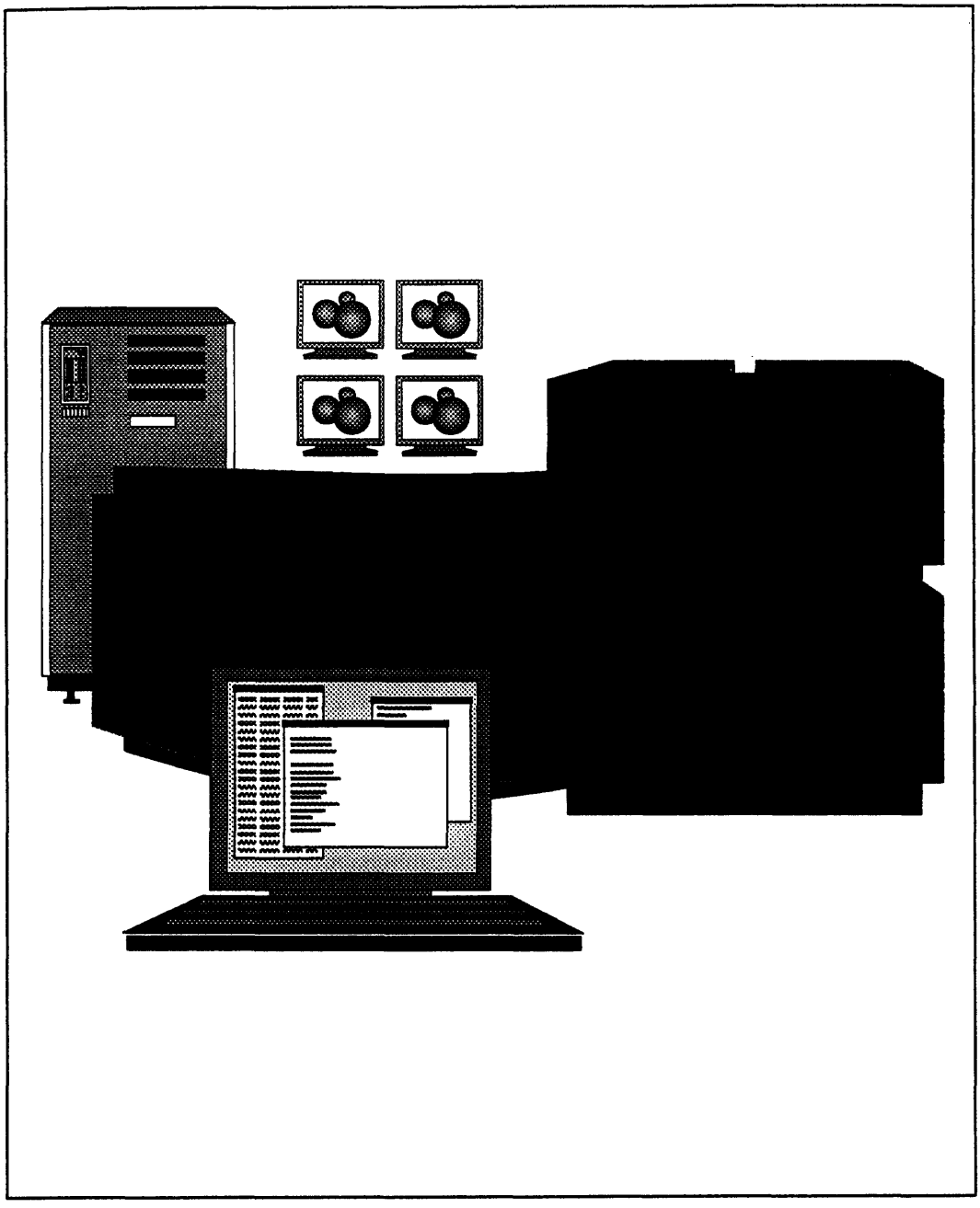


Figure 1. Components of a Connection Machine system. The user's terminal provides access to the front-end computer, CM-2 parallel processing unit, DataVaults, and high-resolution graphics color monitors.

1.3 Data Parallel Computation

Connection Machine systems are designed to operate on large amounts of data. These data sets may be richly interconnected or totally autonomous. A scientific simulation data set, such as a finite-element grid, is highly interconnected, with every node value connected to several element values and vice versa. Disparate values are continually being brought together, computed on, and redispersed. A document data base, on the other hand, may be totally autonomous. The search of any one document proceeds entirely without reference to any of the others. There is no need to combine information from multiple documents in a single computation.

The Connection Machine system is made up of large numbers of processors, each with its own local memory. (Note that a system with 65,536 processors, each with a 128-kilobyte memory, has a total of 8 gigabytes of physical memory.) From the programming perspective, it is possible to think of the memory in either of two ways. When computing on interconnected data sets, it is easiest to think of the memory as a single multi-gigabyte data space. When computing on autonomous data, it is easiest to think of it as many local memories.

Efficient Connection Machine algorithms invariably combine both points of view. When data is being being gathered, it is done in the global context. Once the data is gathered, however, it becomes local data. The ensuing computations are then most easily thought of as being carried out in multiple local memories.

Physical Processors and Memory

The unit of data in a Connection Machine is the parallel variable. A parallel variable is not a new concept; all Fortran arrays, for example, are parallel variables. The array `A(64000)` is a parallel variable with 64,000 individual data elements. The array `D(1000,1000)` is a parallel variable with 1,000,000 data elements. What is important in the Connection Machine system is the way such variables are allocated into physical memory. They are not allocated contiguously in the 8-gigabyte global memory space, because to do so would bunch the variables up in the local memories of the first few processors. Instead, individual arrays are spaced out through the whole address space, so that each processor's local memory space receives the same amount of data. If the number of elements in the array matches the number of physical processors, then each local memory receives one element. If three arrays, such as `A(64000)`, `B(64000)`, and `C(64000)` are defined on a 64K CM-2, then space will be allocated in each local memory for one instance of each variable.

Initialization of these arrays may proceed in either of two ways. The following sequence of Fortran 77 code initializes one element at a time, and hence requires 64,000 units of time:

```
      DO 20 I = 1,64000
20    A(I) = 4
```

An exactly equivalent Fortran 90 statement performs the same initialization in a single unit of time:

```
A = 4
```

A = 4 is a parallel command, and executes for all the elements of **A** (in this case, for 64,000 such elements). On the Connection Machine system, this parallel form is vastly more efficient than its serial counterpart. Initialization of a parallel array to a constant value is carried out by broadcasting the constant from the front end to all the processors at once.

Initialization to a constant is always a local operation. Other computations may or may not be local. An example of a local addition in Fortran would be:

```
      DO 20 I = 1, 64000
20    C(I) = A(I) + B(I)
```

Such a computation proceeds serially, but at each step, the appropriate values of **A** and **B** are to be found in the same local processor, along with the location of **C** into which the sum is to be stored. Such a program sequence also has a vastly more efficient parallel form on the Connection Machine:

```
C = A + B
```

Virtual Processors

The Connection Machine hardware allows each physical processor to operate as many virtual processors, each with a smaller memory. The virtual memory facility is invoked automatically when a parallel variable is declared. Thus the declaration

```
DIMENSION D(1000,1000), E(1000,1000), F(1000,1000)
```

causes the compiler (in this case the Fortran compiler, but other languages have corresponding ways of declaring parallel variables) to invoke a million virtual processors. The

way this invocation is carried out depends on the physical hardware configuration at run time. If there are 65,536 processors available, then each is subdivided 16 ways. (Note that 65,536 times 16 is 1,048,576 virtual processors, slightly more than are needed.) Thus the two statements

```
C = A + B
F = D + E
```

are equally valid parallel statements. Assuming the declarations shown above, the first statement will execute in one unit of time on a 64K CM-2. One unit of time, however, is only enough to do one sixteenth of the second statement. The system automatically cycles fifteen more times to complete the second statement.

Because of the virtual processor capability, these parallel statements also execute on smaller CM-2 or CM-2a systems. On an 8K CM-2a with one gigabyte of memory, the array declaration **A (64000)** invokes 8 virtual processors per physical processor; the declaration **D (1000,1000)** invokes 128 virtual processors. Virtual processors allow CM-2 programs to be completely scalable. They run unchanged on larger and smaller configurations, because the underlying virtual processor configuration changes dynamically to match the code to the hardware resource. When arrays of different sizes are used in the same program (as they typically are) the system changes its virtual processor context as needed to match the current data being operated on.

Global Operations

Global operations are directly supported in hardware just as local operations are. A typical global operation is:

```
X = MAX(A)
```

where **X** is a scalar (as opposed to a parallel) variable. All the values of **A** are compared, and the largest is stored in **X**. This is an example of a *reduction* operation; a large set of values is reduced to a scalar result. Other reduction operations include summation, logical **AND**, and logical **OR**.

Parallel Computation on Interconnected Data Structures

So far we have seen cases where there is no connection between data elements (local computation) and cases where there is total connection between data elements (global reduction). The more typical case lies in between. The inherent structure of most data sets links each data element to some, but not all of the others. Often the linkages are to neighboring elements, so the structure is localized (but not in the absolute sense of "local" used in the previous section). A matrix, for example, is generally thought of as having row and column structure. Elements that have one subscript the same are used in a connected way. If the matrix is used as part of a finite-difference calculation, then the horizontal and vertical neighbors are continually being brought together for computation. If a data structure is converted from the spatial domain to the frequency domain, then a butterfly topology may be invoked during the course of a Fast Fourier Transform (FFT).

It is not possible to arrange interconnected data so that all the pieces of data will reside in the processors that need to use them, because the same piece of data must be used in more than one part of the computation, by more than one processor. Interprocessor communication is required. Computations on data structures have a definite rhythm: first data elements are brought together, then computations are performed. Once the data elements have been brought together, the computations are local. Even on very complex data structures, it is possible to have most of the interacting elements located in the same processor memory. Typically, only a few need to be brought in from another processor's memory.

Localized, Regular Structures

In a localized, regular data structure, the pattern of interconnection is the same everywhere in the data structure, and the paths between interconnected data elements are very short. A finite difference grid is a very typical example. Each data element is connected to its neighbors to the north, south, east, and west. In the three-dimensional case, it is also connected front and back. When the data structure is originally set up, one element of the array is stored in the local memory of each processor.

The CM-2 hardware includes specific communications hardware, the NEWS grid, that supports nearest-neighbor communications on multidimensional rectangular grids. Each Connection Machine language has facilities for moving data along the NEWS grid. A computation on a regular data structure alternates between the gathering of data on the NEWS grid and performing computations locally in the individual processors.

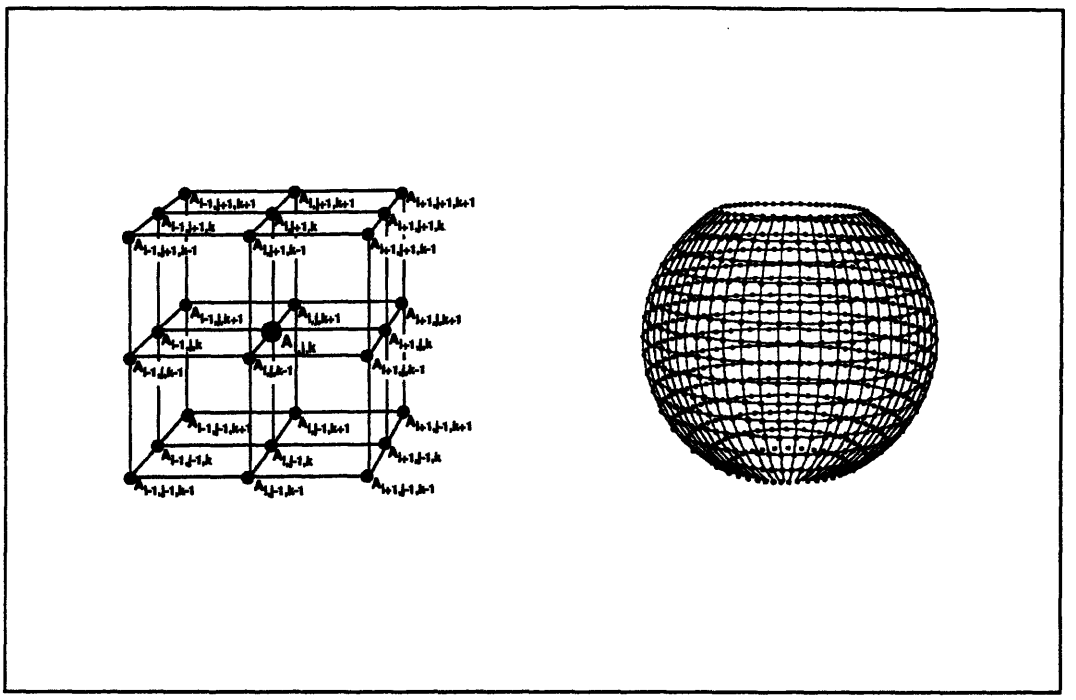


Figure 2. Examples of regular grid structures

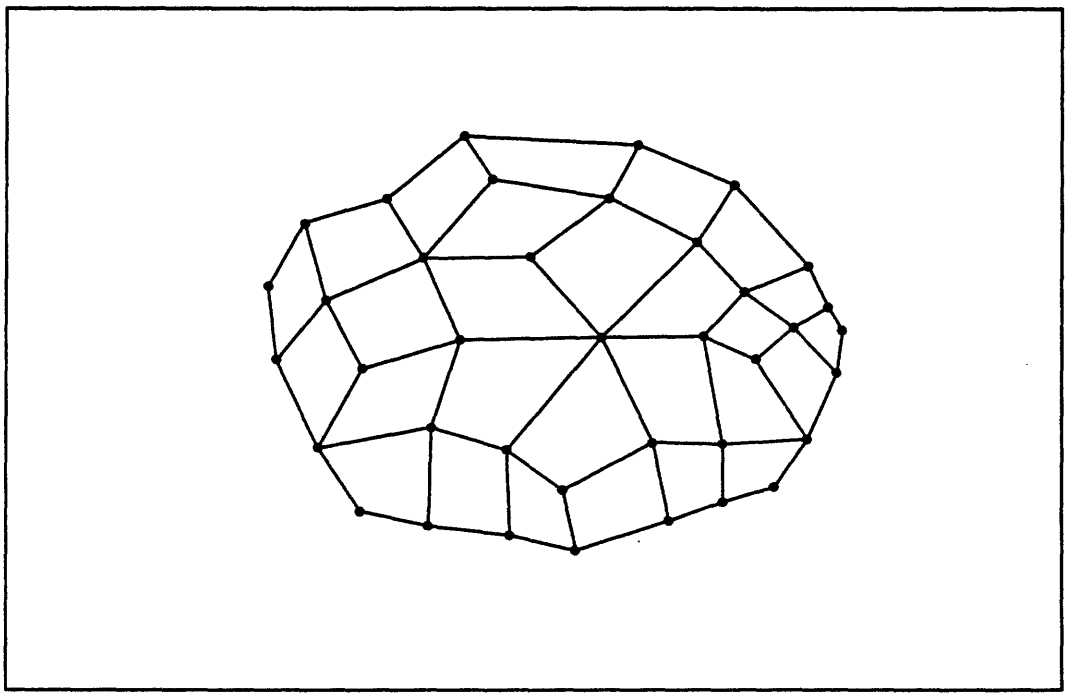


Figure 3. Example of an irregular grid structure

Irregular and Global Structures

In an irregular data structure, the pattern of interconnection varies from one part of the the data to another. Most irregular data structures are relatively localized, but it is also possible to have interconnections between very distant data elements. A finite-element data structure is a common example of a local, irregular pattern. Each of the parallel languages includes language constructs to bring data together and to scatter it back out. Once the appropriate data elements are brought together, the computations on them are carried out in the individual processors.

1.4 Data Parallel Software

To expedite the handling of parallel data structures, Connection Machine system software offers languages that provide data parallel extensions to well-known standards and libraries that facilitate specialized tasks in scientific computing and visualization. It also provides a UNIX-based operating system and various networking capabilities. In all cases, system software is integrated with existing programming languages and environments, so that users can rapidly feel at home on the system.

Fortran on the Connection Machine system is based on Fortran 77 but also uses the array extensions in the draft Fortran 90 standard (proposed by ANSI technical committee X3J3) to express data parallel operations. These array extensions map naturally onto the underlying data parallel hardware.

The C* language, developed by Thinking Machines Corporation, is a data parallel extension of the C programming language (ANSI technical committee X3J11). C* programs are similar in style to C programs; the extensions are unobtrusive and easy to learn. Parallel code looks like serial code, but is executed in all parallel processors simultaneously.

The *Lisp language, also developed by Thinking Machines Corporation, is a data parallel dialect of Common Lisp (a version of Lisp currently being standardized by ANSI technical committee X3J13). *Lisp gives programmers fine control over the CM-2 hardware while maintaining the flexibility of Lisp.

Chapters 2 through 8 of this Technical Summary examine this software environment. Chapters 9 through 14 then discuss the architecture that supports it.

Chapter 2

The Operating System Environment

The Connection Machine Operating System (CM OS) is fully compatible with the UNIX operating system and enhances it in an intuitive manner. The X Window System environment is also fully supported. Since the CM OS follows UNIX standards for its file system and works with UNIX, VME, and HIPPI standards for networking, system managers can integrate their Connection Machine system fully into today's heterogeneous supercomputing environments.

2.1 The Front-End Environment

The front-end computer is the user's gateway to the Connection Machine system. Through the front end, CM users develop, compile, debug, and execute their application programs. Administrators use the front-end computer to configure the Connection Machine system and to execute diagnostic programs for its maintenance; the CM OS offers interactive and automatic features for system administration.

The front-end computer's file system holds all system software and user programs for the Connection Machine system. (Data for user programs, on the other hand, is usually stored in the CM file system, which is discussed later in this chapter.) Programs execute in the front-end environment, with the front end passing instructions to the Connection Machine supercomputer, transmitting I/O requests to its associated I/O and display devices, and receiving program output and messages.

A front end is a multi-user UNIX system: either a Sun-4 Workstation that runs the SunOS operating system and contains a VMEbus, or a Digital Equipment Corporation VAX 8000 or 6300 series minicomputer that runs the ULTRIX operating system and contains a VAXBI I/O bus. Different types of front-end computers may be included in a single Connection

Machine system and may run applications simultaneously on that system. A Connection Machine system can have from one to four front ends.

2.2 Partitioning the Connection Machine System

A user process can access either the complete Connection Machine supercomputer or some portion of it. The portion accessed (whether the whole or the part) is called a partition. A partition is created and allocated for dedicated or multi-user use by a system process or user process giving an **`attach`** command. It is deallocated when the process detaches.

Flexibility for the Administrator

At any given time, a partition of a Connection Machine system can be available as a multi-user partition, a batch partition, or a single-user partition.

- A multi-user partition runs timesharing, and may run batch. Such a partition executes interactive programs for any number of users (up to an optional limit). If the administrator chooses, batch jobs from one or more batch queues may share resources with the interactive jobs, either at the same priority or at a different priority.
- A batch partition runs batch, but not timesharing. This partition is under the control of a batch queue. Interactive jobs may access the partition when no batch jobs are running. The batch queue, however, may be configured so that it detaches such users when the next batch job requires the partition.
- A single-user partition does not run timesharing, but may run batch. This type of partition is available either to one interactive user or to one interactive or batch user at a time.

Changing a partition among single-user, multi-user, and batch modes is as simple as starting or stopping a system process. It can be done interactively, or through scripts set to operate at particular times. Thus, an administrator might choose to divide a 32K system into one 16K multi-user partition and one 16K single-user partition for daytime use, but configure that same system as a single 32K batch partition on nights and weekends, with the changeovers occurring automatically at preset hours.

Flexibility for the User

A system administrator has the option of leaving the entire Connection Machine system in single-user mode. In this case, user programs dynamically partition the CM by attaching to and detaching from those portions of the machine required by their programs.

In other cases, users work within the partitioning established by the system administrator, choosing the multi-user partition or batch queue that best suits their needs.

CM Model	Number of Processors	Possible Partitions
CM-2	64K	16K, 32K, 64K
CM-2	32K	8K, 16K, 32K
CM-2	16K	8K, 16K
CM-2a	8K	8K
CM-2a	4K	4K

2.3 The NQS Batch System

The Connection Machine uses the Network Queueing System (NQS) batch system, which is becoming standard for UNIX networks. This batch system supports two types of queues: batch queues, which are linked to a specific partition, and pipe queues, which feed jobs (via batch queues) to whichever partition is available to run them. It allows submission of batch jobs over the network, either via pipe queues or via `rsh`. It also allows the NQS manager to control the number and characteristics of available queues, as well as each queue's hours of operation.

The hours during which a batch queue executes jobs and the hours during which it accepts jobs are not necessarily identical. For example, a queue might accept jobs from 8 am till midnight, but execute jobs between 8 pm and 8 am. (A queue that accepts jobs is said to be *enabled*; one that executes jobs is said to be *started*.)

Creating and Configuring Queues. An NQS manager decides how many queues to create and what characteristics each queue will have, thus tailoring the batch system to the needs

of the particular site. The administrator uses the `qmgr` utility to create each queue, naming and describing the queue and defining

- the hours during which the queue operates (queues with restricted hours start and stop automatically at designated times)
- the priority of this queue in relation to other queues
- the users or groups of users who can submit jobs to the queue
- time and size limitations for jobs executing from the queue
- the CM system resources available to jobs executing from the queue
- whether the queue has exclusive use of its sequencer(s) and whether it can forcibly detach other users
- whether a queue provides automatic attaches and detaches for its jobs

Submitting Batch Requests. Frequently, the NQS manager defines a number of queues with different characteristics. Users can then choose the queue most suitable for each program. In addition, users can further define the execution environment for a program by using options to the job submittal command that

- request that execution be delayed until a particular time
- request the use of a specified shell
- request that all environment variables be exported with the job
- direct the method by which output is to be handled
- set various per-process limits
- assign a priority to the job

Users can also ask for notification by mail of their job's progress, and can query the system for information on the characteristics and availability of queues and on the status of queued requests.

Controlling Batch Queues. NQS operators can start and stop queues, enable and disable queues, and shut down NQS. When necessary, they can also remove waiting and executing jobs from queues.

2.4 Timesharing

Each multi-user partition is a virtual machine environment, only slightly smaller than the physical environment, operating under the control of a timesharing executive. The executive arbitrates memory demands among user processes, switches the full CM context between processes, and swaps user processes to disk. Swapping to a DataVault — the recommended method — allows a speed of 25 megabytes per second (or 25 megabytes per second per DataVault for striped DataVaults). Administrators can tune timesharing parameters to best meet the needs of their sites; parameters include:

- the maximum number of processes that can execute simultaneously
- the minimum amount of time for which a process can be scheduled
- the maximum amount of memory any process will be allowed to consume
- the desired latency for the scheduler
- the amount of disk space to use for swapping
- the level of logging and the location of the log file

2.5 The Program Development Environment

The program development environment available to Connection Machine users offers the full capabilities of the UNIX and X windows systems. In addition, it offers enhancements specific to CM parallel programming: parallel languages, specialized libraries, and programming tools. It also offers users access to two file systems: the front end's own UNIX file system and the UNIX-based, parallel, high-performance CM file system.

Users write programs in CM languages: C*, CM Fortran, or *Lisp. These languages are supersets of C, Fortran 77, and Common Lisp, respectively. Programs may also include code written in Paris (the Connection Machine's PARallel Instruction Set), and calls to specialized libraries, such as the Connection Machine Scientific Software Library (CMSSL).

Once the program is coded, it is compiled with a CM compiler. (Programs written in *Lisp may be either compiled or interpreted.) Profiling options can be used during compilation, to allow profiling with the `gprof` command.

When a program is compiled, sequential code is translated directly to the native machine code of the front end. Source-level constructs that correspond to Connection Machine (data parallel) operations are translated to a mix of front-end machine code and instructions for the parallel processing unit. In typical programs, data structures are created in the

Connection Machine memory and are used in precisely the same manner as structures in front-end memory. The difference is that operations on the Connection Machine structures can be carried out on many data items in parallel.

Debugging may be done either with a debugger resident on the front end or with **cmdbx**, the CM's parallel extension of the **dbx** debugger.

2.6 The Program Execution Environment

Program execution takes place, as mentioned above, either interactively or in batch mode. In either case, the program executes on the front-end computer and accesses the CM, its I/O subsystem, and its display hardware as needed. The program may access a DataVault, for example, either to read and write data, or to store the results of a checkpointing operation.

Several facilities — checkpointing, timing, and safety-checking — are available to aid program development and robustness during execution. In addition, tools are provided to allow programs executing on single-user partitions to initialize the partition.

Checkpointing

Many applications that run on the Connection Machine system require extended execution times, because of complexity, number of iterations, or both. Users need to be able to interrupt and later restart such a program: perhaps to allow it to run only when the system is not needed for other use, to allow for scheduled machine downtime or protect against unscheduled halts, or simply to allow for restarting the program from some intermediate state during debugging. The Connection Machine system supports this need with a checkpointing facility.

Checkpointing a program lets the user save (and later restart) an executable copy of a program's state. This includes the program's state on the front end, its state on the CM, a list of the files that the program had open at the time of the checkpoint, and a stored copy of the checkpointed program.

The CM checkpointing facility offers three basic methods of checkpointing:

- inserting checkpoints at particular points in a program
- having checkpoints occur periodically
- having a checkpoint occur when a program is sent a particular signal, such as the signal sent during a planned shutdown of the system

Checkpointing can be used from within a debugger, such as **dbx**, and it can be used on programs that execute only on the front end as well as on programs that use the CM.

Timing

A CM timer calculates, with microsecond precision, both the total elapsed front-end time (wall-clock time) and the total amount of time the CM is active. Calls to CM timers can be inserted anywhere in a program. A program can use (and nest) up to 64 timers for simultaneous coarse-grain and fine-grain timing.

Safety Checking

Users may choose to enable a CM safety-checking utility during program execution. This utility checks both for low-level errors and for inconsistencies in programs. Although safety checking reduces execution speed, it can be useful in developing and debugging programs.

2.7 The Connection Machine File System

The Connection Machine File System (CMFS) is designed to handle the immense amounts of data needed by many CM applications, to store data in a format most suitable for the parallel processing done by the CM, and to transform data between this parallel format and the serial format used by other systems.

Serial and Parallel Formats

A serial-format file consists of a single stream of data, suitable for processing by the front end or any other sequential computer. A parallel-format file contains many streams of data, one per Connection Machine virtual processor. It also contains geometry information indicating whether the many streams are to be organized as a multidimensional grid and, if so, how they are to be organized.

CMFS library calls are available to transpose data from serial to parallel format, and vice versa. Thus, data can be exchanged between the Connection Machine system and other systems.

A UNIX-Like File System

The Connection Machine file system is closely modeled on the UNIX file system. It has a hierarchical directory structure, with directories and files identified by pathnames. Many CMFS user commands and library calls have counterparts in the UNIX file system that are similar in name and function. Users will thus find considerable congruence between their front end's file system and the CM file system, and will handle the two in very similar ways.

As in UNIX, any I/O device in the CM I/O system is regarded from the user's perspective as just another file in a file system. I/O to all devices in the Connection Machine system is handled in the same manner, regardless of the type of device involved. For example, writing data from the CM to a DataVault appears to be no different than writing to a tape drive. Users need to learn only a single set of commands and library calls to move data among all devices in the system.

2.8 CM Diagnostics

A complete set of diagnostics for the Connection Machine system is provided with Connection Machine software. Facilities are also provided to make it easy to send error reports and details of diagnostic failures through an electronic message network to the Customer Support Group at Thinking Machines Corporation.

Part II
Parallel Software

Chapter 3

Languages

Data parallel languages allow the programmer to organize data so that program operations may be applied to many elements of data at once. There are few differences in coding style between a data parallel program and a conventional serial program. In both cases a single sequence of instructions is used, with a serial control structure. However, parallel operations are performed on many data items at once. Thus the programming languages for the Connection Machine system provide parallel processing without requiring the programmer to indicate synchronization explicitly in programs.

Because the data parallel and serial programming styles are similar, they can use almost identical languages. The languages currently supported for the Connection Machine system are CM Fortran, C* (pronounced *see-star*), and *Lisp (pronounced *star-lisp*). CM Fortran implements the Fortran 90 array features directly. Each of the other two languages extends its corresponding serial language specification by adding a new data type. Very little new syntax is added; the power of parallelism arises simply from extending the meaning of existing program syntax when applied to parallel data.

CM Fortran and C* are the most commonly used languages for numeric applications. *Lisp is commonly used for artificial intelligence and other symbolic processing applications, but also provides excellent numerical performance with all the convenience and power of the Lisp language.

The following sections survey some broad themes, common to any data parallel programming language, that are useful to keep in mind when examining a language description.

3.1 Establishing Parallel Data Structures

Data parallel programs can be expressed in terms of the same data structures used in serial programs. Emphasis is on the use of large, uniform data structures, such as arrays, whose elements can be processed all at once. A statement such as `A = B + C`, which in a serial language adds a single number `B` to a single number `C` and stores the result in `A`, can equally well indicate thousands of simultaneous addition operations if `A`, `B`, and `C` are declared to be arrays. The underlying paradigm is that every array element is in the memory of a different processor; if the number of array elements exceeds the number of physical hardware processors, a virtual processor mechanism transparently maintains the paradigm.

Although array declarations can thus in principle be identical for serial and data parallel programs, the underlying architecture may require careful allocation of array elements to physical processor memories for best performance. Indeed, in some cases very small arrays are better processed serially. Therefore, data parallel languages, like vector processing languages, generally provide some optional declarations that give the user control over data allocation. In CM Fortran, the compiler automatically determines, according to how the array is used in the program, whether an array should be considered serial or parallel, and how parallel arrays should be allocated to Connection Machine processors. The programmer can override such automatic decisions by inserting declarations in the form of structured comments. In C*, the data types in a declaration implicitly specify whether a data structure is parallel. In *Lisp, data structures are created dynamically, and the programmer uses different allocation operations to specify creation of serial or parallel data structures.

The choice of parallel data structures is perhaps the most important aspect of data parallel programming. Once data has been properly allocated, executable code follows naturally. It is not necessary to use different operation names for different cases. Parallel code can look just like serial code, in the same way that floating-point arithmetic looks like integer arithmetic. A conventional compiler examines the declarations of variables `B` and `C` to determine whether `B + C` will require an integer or floating-point add instruction. In the same way, a compiler for a data parallel language examines declarations to determine whether `B + C` will require a single addition operation or thousands.

3.2 Establishing Linkages among Data Elements

During the execution of a program, data from different problem elements are used together. Data parallel programs use pointers or array subscripts to establish connections between processors and hence between their data elements. An array of pointers, itself a parallel

data structure, establishes an arbitrary pattern of intercommunication. If the required patterns are regular and local, such as processors sharing data with their nearest neighbors, then no explicit array of pointers is needed because each processor can easily calculate the address of its neighbors as needed (in some cases implicitly, with the assistance of special hardware such as the NEWS grid).

3.3 Establishing Scalar Data

Some data is not parallel. For example, it may be wasteful to place a copy of a constant permanently in every processor's memory since the constant can be efficiently broadcast as needed from a central point. For this reason, scalar data (whether constant or variable) may be declared as such and stored in the front end.

3.4 Operations on Mixed Data

Operations that use both scalar and parallel data typically involve replication or reduction. If a scalar value participates in an operation that yields a parallel result (such as adding a constant to every element of an array), the scalar value is replicated by broadcasting it to all processors at once. If parallel data participates in an operation that yields a scalar result, such as finding the sum of all of the elements of an array, a reduction operation is used; given one processor for each data element, such an operation can be completed very quickly by organizing the operations on the data into a balanced binary tree. (This organization is carried out by the underlying language implementation.)

3.5 Conditionals

Data parallel programs implement conditionals by limiting the impact of operations to a certain subset of processors, and hence to a subset of the data elements of a parallel data structure. A conditional operation first tests a specified condition in all elements of a parallel data structure, and then performs the operations only for array elements where the conditional was true. As in serial programs, conditionals may be nested in very general ways.

Chapter 4

Fortran

Fortran for the Connection Machine system is standard Fortran 77 supplemented with the array-processing extensions of the ANSI and ISO (draft) standard Fortran 90. These extensions provide convenient syntax and numerous intrinsic functions for manipulating arrays.

Newly written Fortran programs can use the array extensions to express efficient data parallel algorithms for the CM. These programs will also run on any other system, serial or parallel, that implements Fortran 90. CM Fortran also offers several extensions beyond Fortran 90, such as the **FORALL** statement and some additional intrinsic functions. These features are well known in the Fortran community and are particularly useful in data parallel programming.

4.1 Structuring Parallel Data

Fortran 90 allows an array to be treated either as a set of scalars or as a first-class object. As a set of scalars, array elements must be referenced explicitly in a **DO** construct. In contrast, a reference to an array object is an implicit reference to all its elements (in undefined order). For example, to increment the elements of array **A(100)** by 1, a program can reference the array either way:

A as a set of scalars

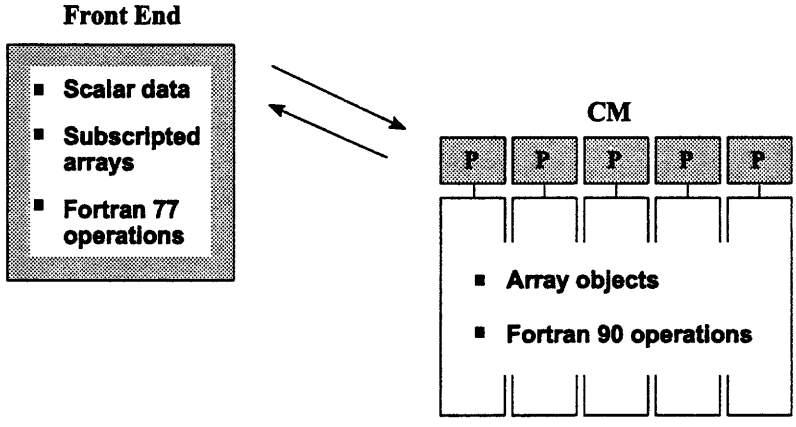
```
DO I=1,100
  A(I) = A(I) + 1
END DO
```

A as an object

```
A = A + 1
```

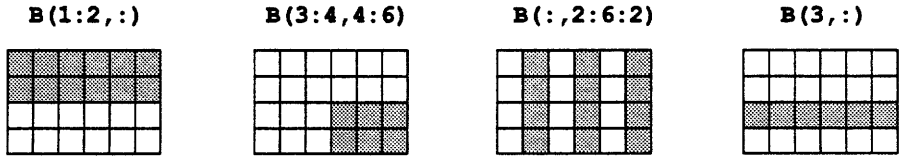
To operate on multidimensional arrays, `DO` loops must be nested to reference each element explicitly. In the statement `A = A + 1`, however, `A` could be a scalar, a vector, a matrix, or a higher-dimensional array.

CM Fortran takes advantage of this standard feature when allocating arrays on the CM system. An array that is used only as a set of scalars is stored and processed on the front-end computer in the normal serial manner. Any array that is referenced as an object is stored in CM memory, one element per processor, and processed in parallel. In essence, the front end executes all of CM Fortran that is Fortran 77, and the CM executes all the array extensions drawn from Fortran 90. No new data structure is required to express parallelism.



The simple array reference `A` is the default form of a *triplet subscript*, `A(1:100:1)`, which resembles the control specification of a `DO` loop. Using triplet subscripts, you can replace `DO` loops with an array reference that indicates all the elements of interest — and thereby cause the array to be processed in parallel on the CM system.

An implicit triplet — that is, the array name alone — is usually used for whole arrays. You can, however, specify any of the control variables, just as in a `DO` loop, to indicate a *section* of the array. For example, some sections of array `B(4,6)` are:



Array sections can be used anywhere that whole arrays are used — in expressions and assignments and as arguments to procedures.

4.2 Computing in Parallel

The most straightforward form of data parallel computing is *elemental* computing, that is, operating on array elements all at the same time, each independently of the others. Any array assignment where the array is referenced as an object has this effect. For example, consider an assignment statement for a 40 x 40 x 40 array **C**:

```
C = C**2
```

The CM system allocates one element of **C** in each of 64,000 processors, and all the processors operate on their respective elements of **C** at the same time.

An expression or assignment can involve any number of arrays or array sections, as long as they are all of the same shape. Scalars can be intermixed freely in array operations, since Fortran 90 specifies that a scalar is effectively replicated to match any array. For example, the following statement assumes that **D** and **E** are 10 x 10 matrices and **F** is a 10 x 100 x 100 array:

```
D = E*2 + 1 + F(:,1:10,3)
```

Another form of array operation uses an *elemental* intrinsic function. Fortran 90 extends most of the intrinsic functions of Fortran 77 so that they can take either a scalar or an array as an argument. If **G** is an array, this statement operates elementally:

```
G = SIN(G)
```

An array assignment can be performed conditionally if it is constrained by a **WHERE** statement. This statement includes a logical mask; it behaves like a **DO** loop with an embedded **IF** statement (except that the order in which elements are processed is undefined). For example, to avoid division by zero in an array assignment, one might say:

```
WHERE (D.NE.0)  E = E/D
```

Finally, CM Fortran offers a form of elemental array assignment, the **FORALL** statement, whose action is position-dependent. The syntax of a **FORALL** statement resembles a **DO** construct, but the assignments can be executed in parallel. For example, to initialize **H** as a Hilbert matrix of size **N**:

```
FORALL (I=1:N, J=1:N)  H(I,J) = 1.0 / REAL( I+J-1 )
```

FORALL can use a mask to make its action dependent on either the value or the position of the individual array elements. For example, to clear matrix **H** below the diagonal, one can set a mask to select those positions where row index **I** is greater than column index **J**:

```
FORALL (I=1:N, J=1:N, I.GT.J )  H(I,J) = 0.0
```

To initialize a table of integer logarithms:

```
FORALL    (I = 1:10)    LG (2**(I - 1): 2**I - 1) = I - 1
```

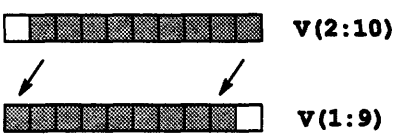
4.3 Communicating in Parallel

A second form of data parallel computing requires processors to access each other's memories, all at the same time. The pattern of interprocessor communication can be either regular (grid-based) or arbitrary. Fortran 90 defines a number of features that move data from one array position to another; these features map naturally onto the communication mechanisms implemented in CM hardware.

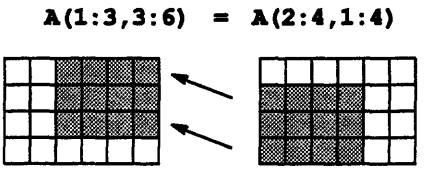
Grid-Based Communication

Many applications, such as convolutions and image rotation, need to move data in regular grid patterns. One way to specify such motion in Fortran 90 is by assigning array sections. For example, to shift vector values to the left:

```
V(1:9) = V(2:10)
```



To shift data on more than one dimension:



Fortran 90 also defines intrinsic functions that perform grid-based data motion. The function **CSHIFT** performs a circular shift of array elements, and **EOSHIFT** performs an end-off shift. For example, the following statement shifts the elements on the second dimension of **A** by one position to the left and assigns the result to **B**. (The **SHIFT** argument can also be an array, which shifts the rows by different offsets.)

```
B = CSHIFT( A, DIM=2, SHIFT=1 )
```

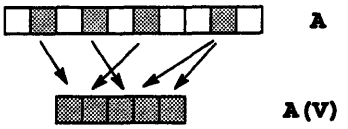
For solving differential equations, CM Fortran programs can access a library of *stencil* routines. These routines combine pipelined floating-point arithmetic with highly optimized grid communications, often in several directions at once. The communication patterns are customized to the various stencil, or star, patterns of the finite difference discretizations of these equations. For example, the two-dimensional, five-point stencil routine gets data from four grid neighbors and performs four adds and five multiplies; it achieves speeds considerably beyond those achieved by manual coding of the same operations.

General Communication

Processors must communicate in arbitrary patterns to map an unstructured problem onto a grid or to index into arbitrary locations of an array. To perform these operations in parallel, CM Fortran provides vector-valued subscripts and **FORALL**.

A vector-valued subscript is a form of array section that uses a vector of index values as a subscript. If **A** is a vector of length 10 and **P** is a permutation of the integers from 1 to 10, then **A = A(P)** applies this permutation to the values in **A**. The statement **A(P) = A** applies the inverse permutation.

The index values can be repeated, which causes element values to be repeated in the section. For example, if **V** is the vector **(/2,6,4,9,9/)**, then **A(V)** is a five-element vector whose values are **A(2)**, **A(6)**, **A(4)**, **A(9)**, and **A(9)**, in that order:



The **FORALL** statement provides the same arbitrary indexing into an array of any rank. For example, the following statement uses the two-dimensional index arrays **X** and **Y** to permute the values of a two-dimensional array **B**:

```
FORALL (I=1:N, J=1:M)  C(I,J) = B( X(I,J), Y(I,J) )
```

4.4 Transforming Parallel Data

Fortran 90 defines a rich set of intrinsic functions that take an array argument and construct a new array (or scalar). All these transformational functions take only array objects (not arrays subscripted in the Fortran 77 manner), and all are therefore computed in parallel on the CM.

One set of transformational functions is the reduction intrinsics, such as **SUM** or **MAXVAL**. These functions apply a combining operator to the elements of an array (or array section) and return the result as a scalar. For example, given a 100 x 500 matrix **D**, the following expression returns the sum of the elements in the upper left quadrant:

```
SUM( D(1:50,1:250) )
```

These functions can take a mask argument to make the reduction conditional. If applied only to a specified dimension, they return an array of rank one less than the argument array. For example, given the 100 x 500 matrix **D**, the following expression returns a 100-element vector containing the sums of the positive elements in each row.

```
SUM( D, DIM=2, MASK=D.GT.0 )
```

A parallel-prefix, or *scan*, operation applies a combining operator cumulatively along a grid dimension, giving each element the combination of itself and all previous elements. These operations, which are useful in such algorithms as line-of-sight and convex-hull, can be expressed with the **FORALL** statement and a reduction function. For example, in the

following add-scan (or sum-prefix) operation, each element of **B** gets the sum of all elements up to and including the corresponding element of **A**:

```
FORALL (I=1:N)  B(I) = SUM( A(1:I) )
```

The array construction functions transform arrays in a wide variety of ways. For example, **TRANSPOSE** performs matrix transposition; **RESHAPE** constructs a new array with the same elements as the argument but a different shape; **PACK** and **UNPACK** behave as gather/scatter operations; and **SPREAD** replicates an array along a new dimension. CM Fortran also provides the Fortran 90 array multiplication functions, **DOTPRODUCT** and **MATMUL**. In addition to the standard Fortran 90 intrinsics, CM Fortran also offers the functions **DIAGONAL**, **REPLICATE**, **RANK**, **PROJECT**, **FIRSTLOC**, and **LASTLOC**.

4.5 To Learn More

To learn more about programming in Fortran on the Connection Machine, see *Getting Started in CM Fortran*, from the Connection Machine documentation set, and *Implementing Fine-Grained Scientific Algorithms on the Connection Machine Supercomputer*, Technical Report #TR 90–1.

Chapter 5

The C* Language

C* is an extension of the C programming language designed to support data parallel programming.

The C* language is based on the standard version of C specified by the American National Standards Institute (ANSI). C programmers will find most aspects of C* code familiar to them. C language constructs such as data types, operators, structures, pointers, and functions are all maintained in C*; new features of ANSI C such as function prototyping are also supported. C* extends C with a small set of new features that allow programmers to use the Connection Machine system efficiently.

C* is well suited for applications that require dynamic behavior, since it allows the size and shape of parallel data to be determined at run time. In addition, it provides programmers with all the standard benefits of C, such as block structure, access to low-level facilities, string manipulation, and recursion. C* also provides a straightforward method for calling Paris functions and CM Fortran subroutines from a C* program, thus allowing access to these languages when appropriate.

5.1 Structuring Parallel Data

In C*, data is allocated on the CM only when it is tagged with a *shape*. A shape is a way of logically configuring parallel data. C* includes a new construct called *left indexing* that is used in declaring a shape. The left index provides the number of dimensions (or *axes*) in the shape and the number of *positions* along each dimension. Positions correspond to processors (or virtual processors) on the CM. For example,

```
shape [256][512]s;
```

declares a shape **s** that is laid out as a 256 x 512 grid on the CM.

This shape is considered to be *fully specified*, since the number of dimensions and positions are provided at compile time. Shapes may also be partially specified or fully unspecified. C* lets the programmer dynamically allocate and specify shapes, thus providing flexibility in the way they can be used.

Once a shape has been fully specified, one can declare *parallel variables* of that shape. Parallel variables have both a standard C data type and a shape. For example, the code

```
shape [16384]t;
int:t parallel_int1, parallel_int2;
float:t parallel_float1;
```

declares three parallel variables of shape **t**; each consists of 16384 *elements*, laid out along one dimension. Parallel variables interact most efficiently when they are of the same shape. In addition to the above method, parallel variables can also be allocated dynamically.

C* also provides parallel versions of arrays and structures. For example, the code

```
shape [16384]t;
int:t parray[16];
```

declares a parallel array, **parray**, which consists of 16 parallel **int**s of shape **t**. The code

```
shape [16384]t;
struct scalar_struct {
   int a;
   float b;
};
struct scalar_struct:t pstruct;
```

declares a parallel structure, **pstruct**, that consists of the standard C structure **scalar_struct** replicated in each of the 16384 positions of shape **t**.

C* includes pointers to both shapes and parallel variables. As in standard C, C* pointers are fast and powerful.

5.2 Computing in Parallel

Parallel Use of Standard C Operators

C* extends the use of standard C operators, through overloading, to apply to parallel data as well as scalar data. For example, if **p1**, **p2**, and **p3** are all parallel variables of the same shape, the statement

```
p3 = p2 + p1;
```

performs a separate addition of the values of **p1** and **p2** *in each position of the shape*, and assigns the result to the element of **p3** in that position. The additions take place in parallel. If **p1** or **p2** were not a parallel variable, it would first be promoted to parallel, with its value replicated in every element. Note that this line of code looks exactly like standard C; the result differs, however, depending on whether the variables are parallel or scalar.

The with and where Statements

C* adds new statements to standard C that allow operations on parallel data.

The **with** statement selects a current shape. In general, parallel variables must be of the current shape before parallel operations can take place on them. For example, code like the following is actually required to perform a parallel addition like the one shown above:

```
shape [16384]t;
int:t p1, p2, p3;

with (t)
   p3 = p2 + p1;
```

C* also adds a **where** statement to restrict the set of positions on which operations are to take place; the positions to be operated on are called *active*. Selecting the active positions of a shape is known as *setting the context*. The **where** statement in the following example ensures that division by 0 is not attempted:

```
with (t)
   where (p1 != 0)
      p3 = p2 / p1;
```

Serial code always executes, no matter what the context.

Programs may contain nested **where** statements; these cumulatively shrink the set of active positions. The context is passed into functions called within the scope of a **where** statement and is correctly reestablished when returning to an outer level as a result of a **break**, **continue**, **goto**, or **return** statement. Note that the context does not affect the flow of control of a program. One can still use standard C statements such as **if** and **while** to manipulate flow of control.

C* extends the standard C **else** statement for use in conjunction with the **where** statement; using **else** after a **where** reverses the set of active positions. The new **everywhere** statement makes all positions active.

New Operators and Data Type

C* adds a few new operators to standard C. For example, the **<?** and **>?** operators are available to obtain the minimum and maximum of two variables (either scalar or parallel).

C* also includes a new single-bit data type called **bool**. Using parallel **bool**s for flags can save space, since memory in the CM may be allocated on bit, rather than byte, boundaries.

Parallel Functions

Functions in C* can pass and return parallel variables and shapes. If it is not known what the current shape will be when the function is called, you can use the new keyword **current** in place of a specific shape name within the function declaration; **current** always means the current shape.

A useful feature of C* is *overloading* of functions. C* allows you to declare more than one version of a function with the same name — for example, one version for scalar data and another for parallel data. The compiler automatically chooses the right version.

5.3 Communicating in Parallel

C* provides two methods of parallel communication: as part of the syntax of the language and via an extensive library of functions. Both allow communication in regular patterns within shapes and in irregular patterns both within and between shapes. Regular communication is faster.

Regular Communication

C* uses the intrinsic function **pcoord** to provide a self-index for a parallel variable along a specified axis of its shape. For example, if **p1** is of a one-dimensional shape with 16384 positions (and the shape is current), **pcoord** initializes **p1** as shown in Figure 4:

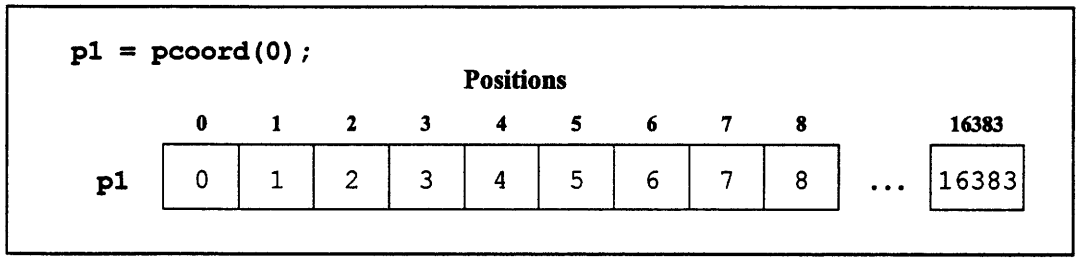


Figure 4. The use of **pcoord** with a one-dimensional shape

The **pcoord** function is typically used to provide regular communication — called *grid communication* in C* — along the axes of a shape. For example, the following code sends values of **source** to the elements of **dest** that are one coordinate higher along axis 0:

```
[pcoord(0) + 1]dest = source;
```

In the common case where **pcoord** is called within a left index expression, and the argument to **pcoord** specifies the axis indexed by the left index, C* allows a shortcut: the call to **pcoord** can be replaced by a period. Thus, for a two-dimensional shape, the following provides grid communication along both axis 0 and axis 1:

```
[. + 1][. - 2]dest = source;        (A chess knight's move)
```

Wrapping from one end of an axis to the other is provided by a standard C* programming idiom that involves the use of **pcoord** along with the new modulus operator **%%** and the **dimof** intrinsic function, which returns the number of positions along an axis of a shape.

Library functions are also available to perform grid communication. For example, the **to_grid_dim** and **to_grid** functions can be used in place of the statements above.

Irregular Communication

C* uses the concept of left indexing to provide communication between different shapes, as well as within-shape communication that does not necessarily occur in regular patterns.

A left index can be applied to a parallel variable. If the index itself is a parallel variable, the result is a rearrangement of the values of the parallel variable being indexed, based on the values in the index. If the index is of one shape and the parallel variable being indexed is of another shape, the result is a remapping of the parallel variable into the shape of the index. Thus, in the following code,

```
dest = [index]source;
```

the parallel variable **dest** gets values from **source**; the values in **index** tell **dest** which element of **source** is to go to which element of **dest**. The variables **dest** and **index** must be of the current shape; **source** can be of any shape. This is known as a *get operation*. Putting the index variable on the left-hand side specifies a *send operation*. Sends are roughly twice as fast as gets. The operations can also be performed with the **send** and **get** functions in the C* communication library. (These are closely related to the Paris operations of the same name.)

5.4 Transforming Parallel Data

C* provides operators and library functions that enable programmers to easily perform common transformations of parallel data.

C* overloads the meaning of several standard C compound assignment operators to provide a succinct way of expressing global reductions of parallel data. For example, **+=**, when applied as a unary operator to a parallel variable, sums the values of all active elements of the parallel variable. The resulting value can be treated the same way as the result of a serial operation. Similarly, the **|=** operator performs a bitwise OR of all elements of a parallel variable. The **reduce** and **global** library functions provide similar capabilities for various operations.

The C* communication library contains many functions that perform other transformations of parallel data. For example:

- The **scan** function calculates running results for various operations on a parallel variable.
- The **spread** function spreads the result of a parallel operation into elements of a parallel variable.
- The **rank** function produces a numerical ranking of the values of parallel variable elements; this ranking can be used to rearrange the elements into sorted order.

Chapter 6

The *Lisp Language

The *Lisp language is an extension of Common Lisp used to program the Connection Machine system in a data parallel style. It allows users to write Connection Machine programs that make full use of the Connection Machine hardware, yet at the same time retain the clarity, expressiveness, and flexibility of Lisp.

The *Lisp language extends the Common Lisp language by providing parallel equivalents for the basic operations of Common Lisp, along with operations that are unique to data parallel programming, such as processor selection, parallel prefix, interprocessor communication, and data shape specification.

A *Lisp program is simply a Common Lisp program that includes calls to *Lisp operators. *Lisp is thus fully compatible with the Common Lisp standard; programs written in Common Lisp will run unmodified in *Lisp. A call to a *Lisp operator on the front-end machine causes all active processors on the CM to execute that operation in parallel. Sequential Common Lisp code, running on the front end, can be freely intermixed with parallel operations; only parallel operations are executed on the CM.

With few exceptions, *Lisp functions and macros are defined via **defun** and **defmacro**, just as in Common Lisp. *Lisp programs are compiled by the Common Lisp compiler, which is extended to recognize and handle *Lisp operations. *Lisp programs can therefore be written, compiled, and tested with the same editors and debuggers as Common Lisp programs.

6.1 Structuring Parallel Data

Scalar and Parallel Data

*Lisp distinguishes between data stored on the front-end computer and data stored in parallel on the CM. Because *Lisp is an extension of Common Lisp, it includes all the standard Common Lisp data types. These data types are all stored in the memory of the front-end machine and are collectively referred to as *scalar* data. *Lisp also supports an additional, parallel, data type called a *pvar*, which is always stored on the Connection Machine. Pvars are collectively referred to as *parallel* data.

A pvar is a *parallel* variable, that is, a single variable with a separate, modifiable value in each processor of the CM. Operations performed on a pvar are performed by all active processors simultaneously, with each processor seeing and modifying only its own value for the pvar. For most of the scalar data types available in *Lisp, there are corresponding pvar data types. The eight basic pvar data types are boolean, integer, floating-point, complex, character, array, structure, and front-end reference.

Creating Pvars in *Lisp

There are three basic ways to create, or *allocate*, a pvar in *Lisp, each designed to serve a specific purpose, as shown in the examples below:

```
(!! 5)                          ;; Allocating a temporary pvar

(*defvar my-five-pvar 5)        ;; Allocating a permanent pvar

(*let ((my-pi!! pi))            ;; Allocating a local pvar
   (*!! 2 my-pi!!))
```

The simplest way to allocate a pvar is via the *Lisp function **!!** (pronounced *bang-bang*), which takes a single scalar value as its argument, and returns a temporary pvar with that value in every processor. For example, the expression **(!! 5)** above returns a pvar with the value 5 in each processor. Most functions in *Lisp operate this way, performing a parallel operation and then returning a temporary pvar value.

One can also allocate a pvar via ***defvar**, which defines a *permanent* pvar, a pvar allocated in such a way that it will remain in existence until it is explicitly *deallocated* by the user. The example above defines a permanent pvar named **my-five-pvar**, and

initializes it to have the value 5 in every processor. (Note that scalar values are automatically promoted to pvars where necessary.)

It is also possible to allocate *local* pvars that exist only for the duration of a body of *Lisp code. By analogy with the Common Lisp operators **let** and **let***, *Lisp includes two operators, ***let** and ***let***, which are used to define these kinds of pvars. The example above defines a local pvar named **my-pi!!**, and then multiplies its value by 2 in every processor.

Defining the Shape of the Data

The shape of the data stored in a pvar is determined by the grid of processors that the CM is currently simulating. The CM can be configured to simulate a grid of processors with up to 31 dimensions, although a one-, two-, or three-dimensional grid is often sufficient. The CM can also simulate more than one grid at a time. The defining property of a processor grid is its *geometry*, which specifies the rank of the simulated grid and the sizes of its dimensions.

The combination of a particular grid geometry and a set of pvars that share that geometry is called a *virtual processor* set (VP set). A VP set is simply a description of a particular abstract configuration of the CM processors, combined with a set of associated pvars that use that configuration. For example, the expression

```
(def-vp-set my-vp-set '(32 64))
```

defines a new VP set named **my-vp-set** with a two-dimensional geometry of 32 x 64 processors.

*Lisp uses the concept of a *current VP set* to determine which VP set is active. Unless otherwise specified, all pvar operations take place within the current VP set. However, you need not concern yourself with defining VP sets if you choose not to. There is a *default VP set* that is automatically defined whenever you start up *Lisp, and until you create and select other VP sets, all pvar operations take place within this default VP set.

*Lisp provides operations to allocate VP sets of many types. The simplest such operator is **def-vp-set**, which allows you not only to define a VP set, but also to specify one or more permanent pvars that will be associated with that VP set. For example, the expression

```
(def-vp-set double-my-vp-set '(64 64)
 :*defvars  ((x 1 nil (unsigned-byte-pvar 8))
            (y 1.0 nil single-float-pvar)))
```

defines a new VP set named **double-my-vp-set** with twice as many processors as in the **my-vp-set** example above. This new VP set has two associated permanent pvars: an unsigned integer pvar **x**, and a single-precision floating-point pvar **y**.

An important feature of the simulated grids defined by VP sets is that they permit the assignment of *addresses* to processors. There are two basic methods used to assign addresses to processors on the CM: *send addressing* and *grid addressing*.

Each processor has a unique numeric *send address* based upon its location within the physical hardware of the CM. The *Lisp operation **(self-address!!)** returns a pvar whose value in each processor is the send address of that processor.

Each processor also has a *grid address*, a sequence of coordinates that defines its position in the *n*-dimensional grid of processors the CM is currently simulating. The *Lisp operation **(self-address-grid!! n)** returns a pvar whose value in each processor is the coordinate of that processor along the **n**th dimension of the current grid.

Accessing and Copying Parallel Data

The most basic pvar operations provided by *Lisp allow you to access pvar values on a per-processor basis, to copy the value of one pvar into another, and to display the elements of a pvar over a range of processors.

The standard function for reading the value of a pvar in a single processor is **pref**. For example, **(pref my-pvar 10)** returns the value of **my-pvar** in processor 10.

The macro ***setf**, analogous to the Common Lisp macro **setf**, is used in combination with **pref** to set the value of a pvar in a specific processor, as in **(*setf (pref my-pvar 10) 123)**, which stores the quantity 123 into processor 10 of **my-pvar**.

It is also possible to refer to processors by grid addresses using the *Lisp operator **grid**, as in **(*setf (pref my-pvar (grid 5 7)) 111)**, which stores the quantity 111 into **my-pvar** at grid location (5,7).

The contents of one pvar may be copied into another by using the assignment operator ***set**. For example, **(*set pvar1 pvar2)** copies the contents of **pvar2** into **pvar1** in all active processors. ***set** is often used to set the value of a pvar in all processors. For example, the statement **(*set pvar1 5)** will store the value 5 into **pvar1** in all active processors.

You can use the *Lisp operation **ppp** (short for **pretty-print-pvar**) to display the values of a pvar. For example, the expression **(ppp (self-address!!) :end 20)** displays the send addresses of the first 20 processors:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

The keyword argument **:end** is used to control how many values are printed. The **ppp** operator provides many keyword arguments that specify the arrangement and format of the pvar values that are displayed.

6.2 Computing in Parallel

The parallel operations supplied by *Lisp are modeled very closely on the existing scalar operations of Common Lisp. *Lisp includes parallel equivalents for most Common Lisp functions and macros. These parallel operations typically have the same name as their scalar Common Lisp counterparts, with either the characters "**!!**" added to the end, or an asterisk "*" appended to the front. The characters "**!!**" are meant to resemble the mathematical symbol ||, which means *parallel*. The asterisk similarly denotes the concept of an operation taking place in parallel. For example, the parallel version of the Common Lisp **mod** function is **mod!!**, and the Common Lisp **if** operator has two *Lisp equivalents, **if!!** and ***if**.

Most *Lisp operators take pvars as arguments and return a pvar result. In general, if a Common Lisp operation takes arguments of a specific data type, the *Lisp equivalent for that operation takes pvars of that data type as arguments and returns an appropriately typed pvar result.

For example, the functions **+!!**, **-!!**, ***!!**, and **/!!** perform the same operations as the Common Lisp functions +, -, *, and /, but take numeric pvars as arguments and perform the appropriate arithmetic operation in parallel. The *Lisp expression **(*set pvar2 (*!! pvar1 pvar2))** multiplies the values of **pvar1** and **pvar2** in all active processors and stores the resulting products in **pvar2**.

*Lisp includes parallel versions of Common Lisp functions for many data types, including operations for complex and character pvars. *Lisp also includes an extensive selection of operators for manipulating array, vector, string, sequence, and structure pvars. There are even operations that allow you to create pvars that reference front-end data structures (such as symbols and lists).

Selection of Active Sets of Processors

Parallel computations can be performed in all processors simultaneously, or in a specific subset of *active* processors selected by the user. Pvar values in inactive processors are always left unmodified. *Lisp provides several macros for selecting the current set of active processors (sometimes referred to as the *currently selected set*).

The most basic processor selection operators are ***when** and ***unless**. Similar to their Common Lisp counterparts, these operators conditionally evaluate a body of code based on the result of a test. The difference is that the test controls which processors will evaluate the body, not whether the body will be evaluated. In the following code sample, ***when** is used to select all processors with odd send addresses. The value of **my-pvar** in those processors is then negated.

```
(*set my-pvar (self-address!!))

(*when (oddp!! (self-address!!))
   (*set my-pvar (-!! my-pvar)))

(ppp my-pvar :end 20)
0 -1 2 -3 4 -5 6 -7 8 -9 10 -11 12 -13 14 -15 16 -17 18 -19
```

The ***all** construct unconditionally selects all processors for the duration of a body of *Lisp code. For example, evaluating the expression **(*all (*set my-pvar 10))** ensures that the value of **my-pvar** in all processors is 10, regardless of the state of the currently selected set.

6.3 Communicating in Parallel

One of the primary strengths of the Connection Machine system is its data communication capabilities. *Lisp includes several kinds of communication operations, including operators for interprocessor communication and for exchange of data between the CM and the front end. These operations are described in the sections below.

Interprocessor Communication

There are two basic kinds of interprocessor communication, regular and irregular.

Regular communication (sometimes referred to as grid, or NEWS, communication) involves a uniform shifting of data across the currently selected grid. The basic *Lisp functions for regular communication on the CM are **news!!** and ***news**. These functions cause each of the active processors on the CM to get or send a value from/to another processor located a fixed distance away across the current grid, with **news!!** performing a parallel get and ***news** performing a parallel send. For example, assuming that the current VP set is two-dimensional, the expression

```
(news!! data-pvar 1 0)
```

will return a copy of **data-pvar** in which each value has been shifted one position along axis 0 of the currently defined grid.

Irregular communication (also known as general router, or scatter/gather, communication) involves a nonuniform exchange of data among processors within that grid. The basic *Lisp functions for irregular communication on the CM are **pref!!** and ***pset**. These functions cause each of the active processors on the CM to get or send a value from/to any processor in the machine, with **pref!!** performing a parallel get and ***pset** performing a parallel send. Both these operations take an address pvar argument whose value in each processor is the address of the processor in the CM with which communication is performed.

If two or more processors attempt to read the data of a single processor, they all receive the same correct data. If two or more processors attempt to write to the same address, the user can specify how they are to be combined (for instance, by summing the values).

In the code sample

```
(*set pvar1 (send-address!!))
(ppp pvar1 :end 9)
0 1 2 3 4 5 6 7 8 9

(*pset :no-collisions pvar2 pvar1
       (-!! *number-of-processors-limit* (self-address!!) 1))

(ppp pvar2 :end 9) ;;; On a 64K machine, the output is:
65535 65534 65533 65532 65531 65530 65529 65528 65527 65526
```

the ***pset** operator is used to "reverse" the contents of **pvar1** (with respect to send-address ordering) and to store the result in **pvar2**.

6.4 Transforming Parallel Data

*Lisp contains many functions to help perform transformations on data. These include operators computing parallel prefixes (scanning) of data, spreading data across the processors of the CM, and sorting and enumeration of pvar values.

Parallel Prefixing (Scanning)

Scanning is a transformation in which a cumulative operation is performed on the values of a pvar across the currently selected grid. There are two main scanning functions in *Lisp, **scan!!** and **segment-set-scan!!**. The extensive options of these functions permit the selection of many kinds of scanning operations, such as addition/multiplication of values; taking the maximum and minimum of values; taking the logical/arithmetic AND, OR, and XOR of values; and even simply copying values across the processor grid.

Scanning can be performed on processors in send or grid address order along any dimension of the currently selected processor grid. Scanning can be performed in either direction along the chosen dimension; the method by which each processor adds its value to the total (inclusively or exclusively) is also controllable.

A simple example of the **scan!!** operation is

```
(ppp (scan!! (self-address!!) '+!!) :end 20)
0 1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190
```

Scanning operations in *Lisp can select segments of processors. Scanning operations are performed independently within segments. This allows the selection of irregular regions of data within which scanning is to be performed.

The **scan!!** operation accepts a segmentation argument for simple uses of this feature. The **segment-set-scan!!** operation uses a special type of pvar, a *segment set* pvar, to allow much finer control over the segmentation of processors than **scan!!** provides.

Spreading, Sorting, and Enumeration

An operation related to scanning is spreading, in which the value of a pvar at a given coordinate along a selected dimension of the currently selected grid is spread to all processors along that dimension. The main *Lisp spreading operator is **spread!!**. *Lisp also includes a related operation that combines the operations of scanning and spreading, called **reduce-and-spread!!**.

The **sort!!** operator reorders the values of a numeric pvar into ascending order. The **enumerate!!** operator assigns to each currently active processor a distinct integer between 0 (inclusive) and the number of active processors (exclusive).

Chapter 7

CM Scientific Software Library

The Connection Machine Scientific Software Library (CMSSL) is a growing set of numerical routines that support computational applications while exploiting the massive parallelism of the Connection Machine system. This library can be linked with code written in CM Fortran, *Lisp, Fortran/Paris, C/Paris, or Lisp/Paris.

The first releases of the CMSSL offer a basic set of linear algebra routines along with Fast Fourier Transforms, random number generators, and histogramming routines. Many of the routines have been implemented to allow parallel computation either on multiple, independent objects (known as *multiple instances*) or on a single large object.

The design of the library addresses the fundamental issues of multiple-object scientific computation on data parallel architectures. In this, the CMSSL offers the foundation for a standard for data parallel algorithms and interfaces.

7.1 CMSSL Capabilities

The library includes many linear algebra routines, two varieties of Fast Fourier Transform (FFT) routines, two kinds of random number generators (RNG), and two histogramming routines for statistical analysis.

Linear Algebra Routines

- *Matrix Multiplication.* Multiplies two real or complex matrices.
- *Matrix Vector Multiplication.* Multiplies a matrix and a vector containing either real or complex data.

- *Vector Matrix Multiplication.* Multiplies a vector and a matrix containing either real or complex data.
- *Matrix Inversion and Linear System Solver.* Inverts a square matrix of real or complex numbers and solves for the values outside the specified matrix. In the high-level languages, these operations are called separately.
- *QR Factorization.* Factors a matrix of real or complex numbers into an orthogonal matrix and an upper triangular matrix.
- *QR Solver.* Given a real or complex matrix decomposed by QR factorization, applies the Householder vectors to the right-hand sides and solves the upper triangular system.
- *Triangular Solver.* Solves a triangular system consisting of the upper or lower triangular portion of a matrix and a right-hand-side matrix, where both contain either real or complex data.
- *Tridiagonal Solver.* Solves one or more tridiagonal systems specified as upper, lower, and diagonal vectors of real or complex data.

All linear algebra routines are designed to support multiple instances. That is, multiple, independent matrices may be solved, transformed, or multiplied concurrently. In addition, where this is relevant, multiple vectors or multiple right-hand sides may be associated with each matrix to be multiplied or solved.

Fast Fourier Transforms (FFTs)

This library routine performs one or more FFT transformations on a multidimensional data set. Interfaces are provided for both simple and detailed versions. The simple version is sufficient for most purposes. The detailed version allows separate specification of the transform direction, scaling factor, and addressing mode along each data dimension and can increase performance for certain cases.

The detailed CMSSL FFT implementation supports performing separate FFTs along each data dimension as well as performing multiple, independent FFTs concurrently along a single dimension.

Random Number Generators

Two lagged-Fibonacci random number generators (RNGs) are provided: the Fast RNG and the VP RNG. The Fast RNG is faster than the standard random number generator included in Paris and the high-level CM languages. The VP RNG produces identical streams on differently sized machines. Both CMSSL RNGs generate pseudo-random numbers as either floating-point or unsigned integer quantities. Both also allow user-controlled reinitialization and checkpointing.

The CMSSL RNGs support multiple instances: multiple streams of random numbers are generated.

Statistical Analysis

Two kinds of histogramming routines are offered: full histogramming and range histogramming. Full histogramming records the distribution of all values within one or more CM data objects. Range histogramming records the distribution of values within specified ranges of values within one or more CM data objects. Both kinds of CMSSL histogram routines can provide an accumulation of totals through successive calls.

7.2 CMSSL Parallel Computation

As a data parallel implementation of familiar numerical routines, the CMSSL represents a set of new solutions to problems of both performance and algorithm choice and design. In most cases, this has led to new implementations of well-known algorithms. The result is a library that brings the power of parallel computation to bear on scientific applications.

Many of the CMSSL routines operate on either single or multiple instances of their operands without requiring additional arguments. For example, matrix vector multiplication may be performed with a single matrix and a single vector by specifying each as an object whose elements are stored one per CM processor. Alternatively, multiple matrix vector products can be computed simultaneously simply by specifying the arguments as a parallel matrix and a parallel vector: one matrix and one vector per CM processor.

In the first case, the single result vector resides in multiple processors; in the second case, each of the multiple result vectors resides in a single processor. In either case the interface is the same. The difference between invoking computation on a single instance and on

multiple instances lies only in the dimensionality of the data structures used as parameters to the particular CMSSL routine.

Consider a second example: the tridiagonal system solver. The parameters to this routine include three vectors that contain the upper, main, and lower diagonals of a tridiagonal system, and a fourth vector that contains the right-hand-side values for the system. Upon completion the solution overwrites the right-hand side.

This one routine interface supports four different degrees of computational concurrency: A single system may be solved. A single system may be solved for multiple right-hand sides. Multiple systems may be solved for a single right-hand side each. Multiple systems may be solved, each for multiple right-hand sides. Using an odd-even cyclic reduction algorithm one or more systems of tridiagonal equations of the form $Ax = b$ are solved.

To solve a single system, specify the upper, main, and lower arguments as one-dimensional (see Figure 5).

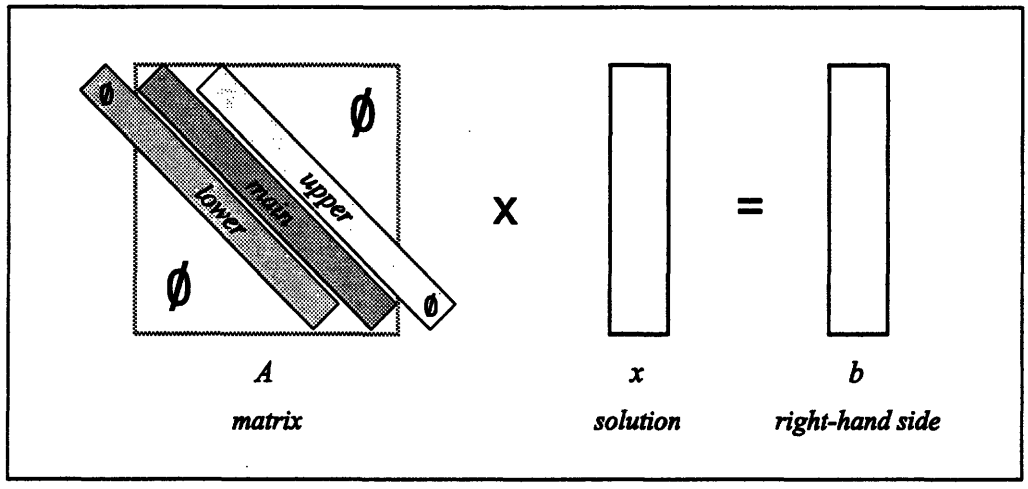


Figure 5. A single tridiagonal system with a single right-hand side

To solve for multiple right-hand sides, the right-hand-side argument (also the solution) is given an in-processor (serial) dimension equal to the number of right-hand sides (*nrhs*) (see Figure 6).

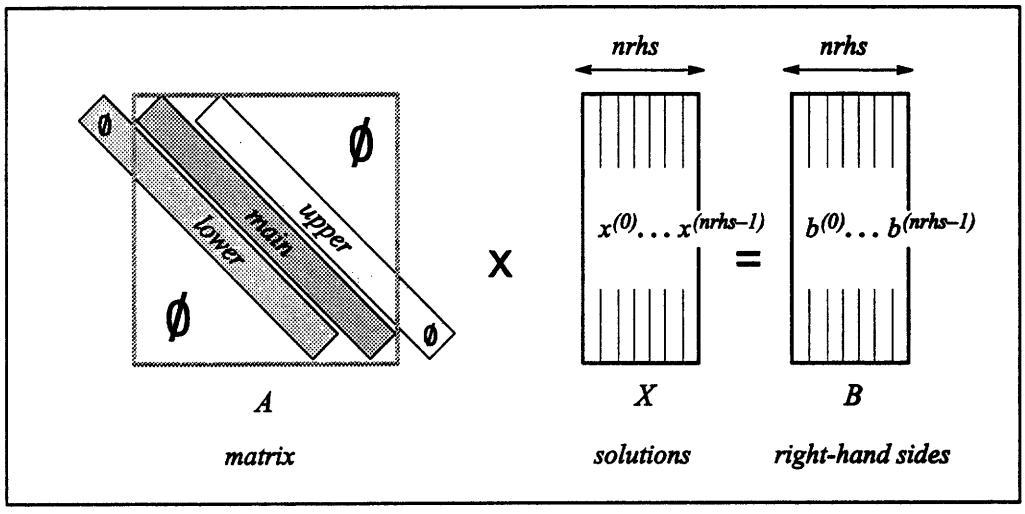


Figure 6. Single tridiagonal system with multiple right-hand sides and solutions

To solve multiple systems, specify the upper, main, and lower arguments with two dimensions: one for the coefficients of the system and one to specify how many systems are represented. The right-hand side (solution) argument must be similarly specified in two dimensions (see Figure 7).

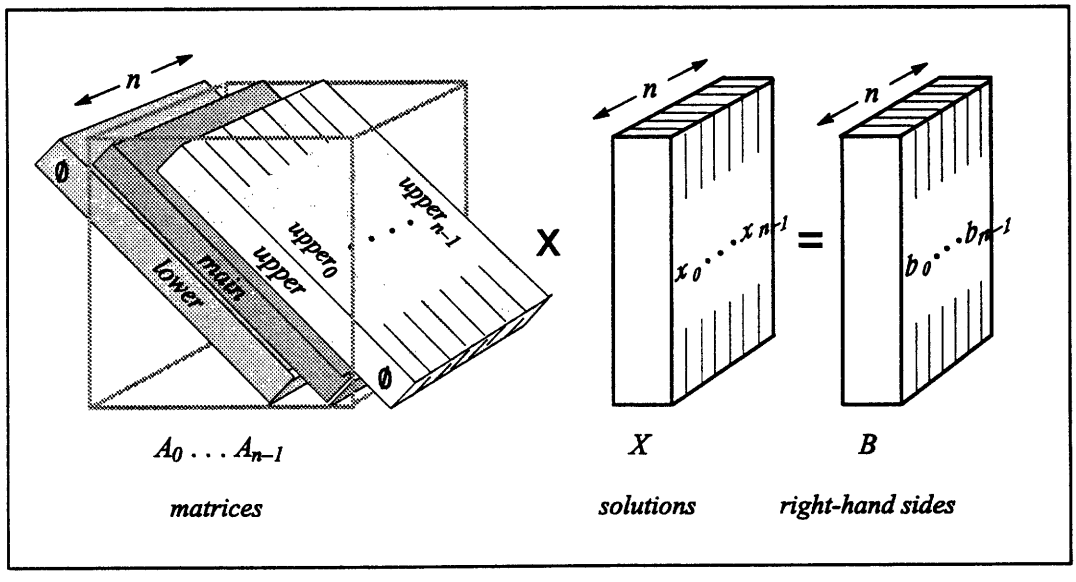


Figure 7. Multiple tridiagonal systems with single right-hand side for each system

To solve multiple systems each with multiple right-hand sides, the right-hand-side (solution) argument is specified in three dimensions: one is the length of the vector and along this dimension lie the right-hand values; one is the number of systems (*n*); and one is the number of right-hand sides (*nrhs*) per system (see Figure 8).

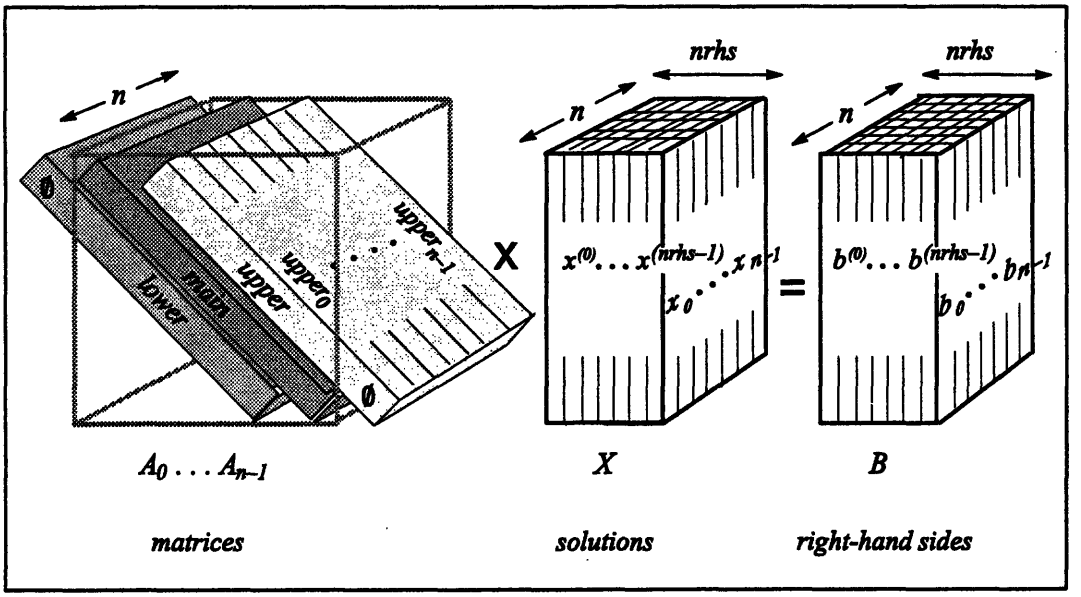


Figure 8. Multiple tridiagonal systems with multiple right-hand sides for each system

The benefit of using CMSSL routines to solve a single instance of a linear problem lies in the speed gained by exploiting the parallel architecture of the Connection Machine system. Computations on matrices require numerous repetitive calculations along one or both axes. On a serial machine, these must be done one at a time, but on a parallel machine they can be done all at once.

Using the CMSSL to solve multiple instances of a linear problem offers similar, but perhaps greater, benefits. For applications that require solving many systems or decomposing many matrices, it is no longer necessary to iterate over the set of systems; the solutions can be computed concurrently.

7.3 Linear Algebra

The CMSSL provides separate, optimized operations to do matrix multiplication, matrix vector multiplication, and vector matrix multiplication. The CMSSL matrix multiplication

implementation uses a systolic algorithm known to be numerically stable. The matrix vector and vector matrix implementations use a Saxpy algorithm also known to be stable.

Similarly, the CMSSL provides separate linear system solving operations for dense matrices and for banded matrices.

Solving Dense Systems

For dense matrices, the CMSSL offers two methods of solving a linear system represented as a real or complex general matrix. The first method uses the Gauss-Jordan linear system solver. The second method uses the QR factorization operation in combination with either the QR solver or the triangular solver. These operations are based on two different algorithms.

The CMSSL linear system solver is based on a variant of the Gauss-Jordan algorithm. The Gauss-Jordan algorithm is known, in some cases, to give residuals that are higher than those resulting from the Gaussian elimination method — by as much as the order of the condition number of the linear system. However, in the CMSSL a variant known as "the rehabilitated Gauss-Jordan algorithm" is implemented and, for well-conditioned systems, this yields results as good as those produced by Gaussian elimination.

Both Gaussian elimination and Gauss-Jordan require pivoting if the system is not symmetric positive definite. The CMSSL linear system solver supports two pivoting strategies: a variant of partial pivoting, where the pivot element is chosen from the pivot *row*, and columns are (in effect) permuted, and conventional total pivoting, where the pivot element is chosen from a submatrix and both rows and columns are permuted. The total pivoting strategy is numerically more stable but slower than the partial pivoting strategy.

Matrix inversion is also accomplished using the same variant of the Gauss-Jordan algorithm. On well-conditioned matrices, this algorithm produces numerically stable matrix inversion results. On ill-conditioned matrices, it fails about as often as LU decomposition.

The second method of solving a dense linear system uses the QR factorization and the QR solver operations. Given one or more systems of the form $AX = B$, this method uses QR factorization to decompose $A = QR$. Next, the QR solver applies the Householder vectors in Q to the right-hand sides in B and then solves the upper triangular system R, while overwriting B with the least squares solution of the linear system. This algorithm produces numerically stable results on well-conditioned matrices.

Solving Banded Systems

Banded linear algebra operations solve systems of equations in which the coefficient matrix has non-zero matrix elements in a narrow band around the diagonal.

A tridiagonal solver provides this functionality in the CMSSL. For diagonally dominant and positive definite systems, the CMSSL implementation of the tridiagonal solver is known to be unconditionally stable. However, for poorly conditioned systems, the algorithm may be unstable. A stabilization strategy is currently planned.

7.4 Fast Fourier Transforms

Continuous physical quantities, such as waves and periodic vibrations, can be represented as summations of sinusoidal components over a range of frequencies. The derivation and manipulation of these frequency series is known as Fourier analysis. The Fourier transform of a function over time or space specifies the amplitudes and phases of each frequency component. Usually this information is expressed as the complex exponential (cos + i sin) of certain harmonics of a fundamental frequency.

Given such a function, the Fourier transform can be used to convert between the time or space domain and the frequency domain. Most applications of the Fourier transform begin as quantities specified or measured over space or time, so the transform of these values into the frequency domain is called a *forward* transform. An *inverse* transform converts frequency-domain values back into time- or spatial-domain values.

The Discrete Fourier Transform (DFT) is the Fourier transform most suitable for numeric work. Its most common implementation is the Cooley-Tukey Fast Fourier Transform or FFT.

A Fast Fourier Transform algorithm is a method of performing the Discrete Fourier Transform, which determines the discrete frequency components of a continuous but discretely sampled complex variable. An FFT is considered fast because it exhibits $O(N \log N)$ complexity, where N is the length of the input sequence. By comparison, a straightforward evaluation of the DFT formula exhibits $O(N^2)$ complexity.

The CMSSL FFT implements an algorithm known as the Radix-2 Cooley-Tukey FFT. The Connection Machine system lends itself well to this particular algorithm, which combines two data elements at each step in a butterfly communication pattern. A butterfly communication pattern always operates between data elements at a distance of 2^N. On the CM, this

is readily done with a very efficient kind of interprocessor communication known as cube wire communication.

FFTs have a wide range of scientific and engineering applications including digital filtering of discrete signals, smoothing and decomposition of optical images, correlation and auto-correlation of data series, numerical solution of partial differential equations such as Poisson's equation, and polynomial multiplication.

The CMSSL provides a complex-to-complex FFT routine with two user interfaces:

- the Simple FFT, used to transform a dataset in the same direction along all axes
- the Detailed FFT, used for all other cases

The FFT is traditionally defined as a one-dimensional algorithm. However, a multidimensional FFT can be done by performing FFTs along each row and column of a grid. The CMSSL FFT operations support *n*-dimensional FFTs, subject to the limits of Paris (31 dimensions) and of machine memory (27 nontrivial dimensions on a CM-2 with 8 gigabytes of memory).

7.5 Random Number Generators

Two varieties of random number generators (RNG) are included in the CMSSL:

- the Fast RNG
- the VP RNG

These random number generators use a lagged-Fibonacci algorithm. They supplement the standard Paris random number generator, which uses a cellular automaton algorithm.

The lagged-Fibonacci algorithm used by both CMSSL random number generators is widely used to produce a uniform distribution of random values. This implementation has been subjected to a battery of statistical tests, both on the stream of values within each processor and for cross-processor correlation. The only test that the CMSSL RNGs fail is the Birthday Spacings Test, as predicted by Marsaglia. Despite this failure, these lagged-Fibonacci RNGs are recommended for the most rigorous applications, such as Monte Carlo simulations of lattice gases.

To construct pseudo-random values, the CMSSL random number generators use *state tables* loaded from the Paris random number generator. The difference between the Fast RNG and the VP RNG lies in the allocation of their state table fields. The Fast RNG allocates one state table per physical Connection Machine processor. The VP RNG allocates one state table per

virtual processor. The Fast RNG (so named because it is much faster than the Paris RNG) thus consumes substantially less CM memory than the VP RNG. The VP RNG can produce identical results on differently sized Connection Machines.

Either CMSSL RNG may be reinitialized for reproducible results and checkpointed to guard against forced interruption.

7.6 Statistical Analysis

The CMSSL statistical analysis routines currently include two histogramming operations. Histograms provide a statistical mechanism for simplifying data. They are generally used in applications that need to display or extract summary information, especially in cases when the raw data sets are too large to fit into the Connection Machine system. Two routines are provided: one that tallies the occurrences of each value in a CM array, and one that counts the occurrences of values within specified value ranges. For particularly large data sets, the range histogram operation facilitates breaking data down into subranges, perhaps as a preliminary step before doing more detailed analysis of interesting areas.

Histograms have many applications in image analysis and computer vision. For example, a technique known as histogram equalization computes a histogram of pixel intensity values in an image and uses it to rescale the original picture.

The CMSSL histogram operations treat the elements of a front-end array as a series of *bins*. In each bin a tally of CM field values or value ranges is stored. The number of histogram bins varies widely with the application, from a dozen tallies on a large process or a few dozen markers on a probability distribution to a few hundred intensity values in an image or a few thousand instruction codes in a performance analysis.

Chapter 8

Data Visualization

Visualization, the graphic representation of data, has come to be an essential component of scientific computing. Visualization techniques range from a simple plotting of data points to sophisticated interactive simulations, but all allow researchers to analyze the results of their computations visually. One can literally "look at" the data to identify special areas of interest, anomalies, or errors that may not be apparent when scanning raw numbers. Visualization is often the only effective way to interpret the large data sets and complex problems common to the applications run on the Connection Machine system.

Thinking Machines Corporation supports visualization on the Connection Machine system through a Generic Display Interface, which permits both image display and user interactions on either an X windows workstation or a dedicated high-resolution graphics display system. Subroutine libraries perform basic visualization tasks to simplify the creation, display, and storage of CM images.

There are three main visualization libraries:

- **Render* provides basic functions for creating and manipulating an image in CM memory.
- The *Generic Display Interface* provides a single interface for displaying image data in CM memory on any X windows display or through the CM graphics display system. Through this interface a programmer can initialize a display, control display parameters, transfer an image to the display, and implement interactivity through a mouse interaction system and text display.
- The *Image File Interface* provides support for storing images from CM memory in TIFF (Tag Image File Format) image files. TIFF is a standard image file format that is widely supported. Functions are also provided that read TIFF files into CM memory, into a front-end array, or directly to a Generic Display.

These libraries may be called directly from CM Fortran, C*, Fortran/Paris, C/Paris, or Lisp/ Paris programs.

8.1 Visualization Output from the CM System

The CM supports three methods for graphic output: the X Window System, the graphics display system, and networked output to graphics workstations.

The X Window System

Thinking Machines supports the X Window System Version 11 interface by supplying "client" extensions that display Connection Machine images on any display running the X Window System. Through the Generic Display Interface, described in more detail below, you can easily establish any X windows server as the display for your visualization programs.

The X Window System has become the dominant vendor-independent workstation window interface standard. It logically separates the support of a window-based display from the computer actually running an application. The display computer may be connected to the application computer by means of a network. Thus, by using the X Window System, users can display the results of their computation on the Connection Machine system on a remote workstation anywhere on the network. X interfaces are now available for most workstations, including those manufactured by Sun Microsystems and Digital Equipment Corporation.

The CM Graphics Display System

In addition to the X windows interface, the Connection Machine system also includes a high-resolution graphics display system consisting of a framebuffer and a high-resolution 19-inch monitor. As explained in Chapter 14, the graphics display system is directly coupled with the parallel processing unit. Because of this tight integration, the CM graphics display system is capable of real-time output.

Geometric Output to Graphics Workstations

For especially demanding visualization applications, software is available that makes it possible to send geometric descriptions of an image from the Connection Machine system to Stardent or Silicon Graphics graphic workstations for rendering and display.

Stardent and Silicon Graphics workstations are specialized graphics-processing workstations that implement many advanced rendering techniques in hardware and offer extensive rendering environments. The CM interfaces to these workstations allow the user to use the Connection Machine system to compute the image geometry (for example, polygon coordinates and color information) and then send it from the CM I/O channel over the VMEbus directly to local memory on the graphics workstation. The image data is sent in a format that may be immediately used by the workstation's specialized graphics processors and software to produce full-color shaded images.

Thus, these interfaces make it possible to establish a high-speed interactive distributed graphics environment in which the Connection Machine system is used to perform the numeric computing that produces the geometric data, and the specialized graphics workstations perform the image rendering that produces the final data visualization.

8.2 *Render

*Render provides basic functions for creating and manipulating an image in CM memory. These functions include

- drawing routines for basic graphics primitives (points, lines, arrays, and spheres)
- graphics math utilities for creating, manipulating, and transforming coordinate vectors and matrices, and for converting between different color spaces
- image processing (dithering) routines that can be used to convert RGB images to grayscale and grayscale to monochrome.

This library is intended as a building block for more advanced visualization tools.

Drawing Routines

*Render draws graphics primitives into an *image buffer* in CM memory. An image buffer is simply a parallel data structure (an array in CM Fortran; a parallel variable in C*) that

is used to collect pixel values for display. The image buffer is organized as a two-dimensional array with one data element for each pixel. Each data element contains the color value and, if the image is three-dimensional, the z-coordinate of the corresponding pixel in the image to be displayed.

The *Render drawing routines allow the user to draw graphic primitives including points, lines, image arrays, and spheres into the image buffer by specifying image buffer coordinates in scalar or parallel variables. When a set of coordinates is specified in a parallel variable, *Render operates on all the coordinate values at once. For example, the line drawing function accepts an array of line coordinates (with each data point containing the starting and ending points for one line) and draws the lines in parallel into the image buffer.

Graphics Math Utilities

*Render provides a toolbox of math routines to simplify basic operations common to computer graphics.

In computer graphics, coordinates are commonly represented as position vectors in coordinate matrices. Transformations of these coordinates (scaling, translation, or rotation) are performed by applying a transformation matrix to the coordinates using the rules of matrix multiplication.

*Render routines create and manipulate coordinate vectors and matrices, and create and combine transformation matrices to scale, rotate, or translate coordinate data. In addition, routines are available that create viewing matrices. These matrices can be used to apply foreshortening, oblique, orthographic, or perspective projections to an image geometry. Parallel versions of these routines are provided that can apply a transformation matrix to an entire array of coordinates in a single operation.

Similarly, colors are often represented as vectors in a "color space." The most common of these is the RGB color space supported by the CM framebuffer. However, different graphics environments may use other color models. *Render provides routines to convert color values from one color space to another. Specifically, routines are available to convert between RGB, CMY, YIQ, HSV(HSB), and HSL.

Image Processing (Dithering)

The color capabilities of a programmer's workstation may range from full 24-bit color, as on the CM framebuffer, to a 1-bit black and white display. *Render routines make it possible to compute your image for color and then convert it for display on grayscale or black and white monitors.

Specifically, an RGB image can be converted to a single-value grayscale and then "dithered" to a 1-bit black and white image. Several different methods, including dot diffusion and error propagation of different orders, can be applied to preserve as much image detail as possible while reducing the number of color intensities used to represent the image.

8.3 Generic Display Interface

The Generic Display Interface provides a single high-level interface that simplifies the display of data from an image buffer. With just two function calls the user can select and initialize a display screen and transfer image data to it from a parallel data structure in CM memory. Currently the display may be an X11 window on any workstation screen, or the CM graphics display system. In addition, support is provided for mouse interactions, including cursor tracking and selection, and for displaying text strings on either display type.

Creating a Display

A single Generic Display routine creates and initializes a display. When calling this routine, the user specifies the width, height, and bits per pixel desired for the display. (The display's bits per pixel, also called the display's *depth*, is the number of bits of color information maintained for each pixel.) The Generic Display Interface then returns a menu containing the framebuffer displays available from the attached Connection Machine and an X windows option.

Once a display is selected from the menu, the Generic Display Interface initializes the display as the *current display*, matching the desired width, height, and bits per pixel as closely as possible. For an X windows host, the interface opens an X window on that server as the current display. For a CM framebuffer, that framebuffer and its attached monitor are initialized as the current display. All the operations of the other Generic Display Interface

routines, including image transfer, display parameter changes, and display information queries, are directed to the currently selected display.

Because the display is selected from a menu at execution time, it is possible to move the image display between an X windows display and a CM framebuffer without changing the program. This simplicity gives a great deal of flexibility during application development. One might view data through an X11 window when a CM framebuffer is not available, preview an image in X11 during development and then switch to the CM framebuffer for final revisions and viewing, or display the image on a remote X server.

Displaying an Image

Transferring an image to the currently selected display also requires only a single call to a Generic Display Interface routine. If no offsets are applied, the interface writes the color value at position (0,0) in the image buffer to the origin (the upper left corner of the display space); each subsequent position in the image buffer supplies the color value for the pixel at the corresponding location in the display space. If the image is smaller than the display, the portion of the display to the right and below the image is left unchanged. If the image is larger, the portion of the image beyond the boundaries of the display is clipped. The Generic Display Interface also allows you to offset the image in the display space.

If the current display is an X windows display, the dimensions of the display window will be as close as possible to the desired width and height you specified when creating the display. The other capabilities of the display, for instance the number of bits per pixel, depend on the capabilities of the workstation supporting the X server you named.

If a CM framebuffer is the current display, the Generic Display Interface transfers the image buffer data from CM memory directly to the framebuffer display memory. Image data written to the framebuffer is immediately displayed on the color monitor. The maximum size of the image that may be displayed is determined by the resolution of the monitor. Normally, this is a high-resolution RGB monitor with a resolution of 1280 x 1024 pixels. The framebuffer also supports an NTSC signal that can be fed into videotape recorders and other standard video processing equipment. The resolution of an NTSC monitor is usually 640 x 480 pixels.

Interactivity

User interaction with the display is supported through the Generic Display Interface's mouse interaction system. These routines allow applications to track mouse movement

with a cursor on either an X windows display or a CM framebuffer and to respond to button presses on the mouse.

Routines are provided to select the mouse host; get the location of points, lines, and boxes defined by the cursor on the display; get a stream of mouse motion events; "grab" the mouse and poll it for button presses; move the cursor; and change the shape and visibility of the cursor.

Text Display

The Generic Display Interface also provides a group of Generic Text routines that support the use of text strings to annotate visualization images. With these routines, one can display text strings on either an X windows display or a CM framebuffer, or draw text strings into an image buffer in CM memory. The strings are positioned by specifying display or image coordinates.

The Generic Text software includes two fonts that are always available. In addition, users working on an X windows workstation may use any of the X11 fonts available on it.

8.4 Image File Interface

The Image File Interface allows image data from the CM system to be stored in files for later display or processing. The interface currently writes the files in the TIFF image file format. This format is supported by a public domain library and is accepted by many other graphics systems and software packages on platforms ranging from personal computers to supercomputers. Thus, this interface makes it possible for you to move images between the CM system and numerous other graphics environments.

Routines are also provided that read the most common classes of TIFF files into an image buffer in CM memory, into an array on the front-end computer, or directly to a Generic Display.

Part III
Parallel Architecture

Chapter 9

Paris

Paris is the PARallel Instruction Set for programming the Connection Machine system.

Used primarily as a base for the higher-level languages of the Connection Machine system, Paris provides a large number of operations similar to the machine-level instruction set of an ordinary computer. Paris supports primitive operations on signed and unsigned integers, floating-point numbers, and complex numbers as well as communication operations and facilities for transferring data between the Connection Machine processors and the front-end computer.

The Paris user interface consists of a set of functions, macros, and variables to be called from compiled code. Where appropriate, users may insert Paris code directly into their programs. Several different versions of the user interface are provided: one for the Fortran programming language, one for C, and one for Lisp. These interfaces are functionally identical; they differ only in conforming to language-specific syntax and data types.

Since Paris is a lower-level language than CM Fortran, C*, or *Lisp, it provides a useful way to understand the Connection Machine architecture. It is presented here to help fill in the picture of the programming model for the CM-2.

9.1 Virtual Machine Architecture

An important property of the Connection Machine architecture is *scalability*. Connection Machine systems can have from 4,096 to 65,536 physical (hardware) processors. In most cases the same software can be executed unchanged on Connection Machine systems with different numbers of physical processors. Using twice as many physical processors, a problem will run in half the time.

Paris enhances this scalability by presenting the user an abstract version of the Connection Machine hardware. The most important feature of this paradigm is the *virtual processor* facility, whereby each physical processor simulates some number of virtual processors. A program can be written assuming any appropriate number of processors; these virtual processors are then mapped onto physical processors. In this way a program can be executed unchanged on Connection Machine systems with different numbers of physical processors. There is an approximately linear trade-off between number of physical processors and execution time. There is a memory trade-off as well: the memory of a physical processor is divided among the virtual processors it supports.

When a Paris **add** instruction is executed, each physical processor may perform many addition operations, one for each virtual processor that is mapped onto that physical processor. Paris also provides virtual-processor versions of the three hardware-supported communications mechanisms: routing, NEWS grids, and scanning.

Virtual Processors and Virtual Processor Sets

The Paris virtual processor mechanism supports data parallel programming by associating one virtual processor, or VP, with each element of a data set. The set of all virtual processors associated with a data set is called a *virtual processor set*, or *VP set*. Consider, for example, an image-processing problem that deals with an image of 65,536 pixels, shaped in a 512 x 128 rectangle. Each pixel is an element of the data set that makes up the image. Thus we would write a program using one VP set of size 65,536: one VP for each pixel. We would configure the NEWS grid for this VP set to be two-dimensional, with shape 512 x 128.

Because a single problem may be composed of more than one data set, Paris allows the simultaneous existence of more than one VP set. For example, a text retrieval program may implement some operations that work with articles and some that manipulate words. This is most conveniently modeled with two VP sets, the first corresponding to the data set of all articles (one VP per article) and the second corresponding to the data set of all words (one VP per word). The second VP set will be much larger than the first.

VP sets are created and deleted through function calls to Paris. The number of virtual processors in the VP set (the VP set *size*) is fixed at the time of the VP set's creation. The VP set size must be a multiple of the number of physical processors. The organization of the virtual processors as a NEWS grid is called a *geometry*. The geometry of a VP set is specified when the VP set is created and may have from 1 to 31 dimensions. While the total number of virtual processors in a VP set remains fixed, its geometry may be altered at any time.

Although multiple VP sets may coexist, only one VP set may be active at any time. This VP set is known as the *current VP set.* All VP sets other than the current VP set are latent; they can not execute any instructions. Paris provides a function for making any specified VP set current. Instructions are then executed in that VP set until some other VP set is made current.

Mapping VP Sets to the Physical Machine

When a Paris program runs, the virtual processors in the user's program are mapped onto the machine's physical processors. The sizes of the VP sets and the size of the physical machine determine how many virtual processors are assigned to each physical processor. In effect, each Connection Machine processor and its memory are shared among the virtual processors they support. These concepts are explained further in the following sections.

VP Ratios

Each virtual processor set has a *virtual processor ratio* (*VP ratio*). The VP ratio indicates how many times each physical processor must perform a certain task in order to simulate the appropriate number of virtual processors. The VP ratio may change when a new VP set is made current.

When the machine is operating within a particular VP set, each Paris instruction is executed many times in each physical processor, once for every virtual processor. This is completely transparent to the user.

The method of assigning virtual processors to physical processors "spreads out" a VP set as much as possible; the VP ratio for each VP set is as low as possible. The burden of handling a VP set is shared equally by all physical processors. If a larger (physical) machine is used on a particular application, the VP ratio required for each VP set is smaller.

This description of "execute once for each virtual processor" applies most accurately to operations such as arithmetic that can take place within each virtual processor independently of other virtual processors. Operations that perform communication are more complicated, but the idea is the same: each physical processor performs all necessary execution steps on behalf of each virtual processor that is to participate in the operation.

As far as the user is concerned, physical processors are hardly visible. Paris is designed to allow the programmer to think entirely in terms of the virtual processor as the basic unit of computation.

Fields

At the time of its creation, a VP set has no associated memory. Paris provides functions to allocate and deallocate memory for a VP set.

Memory is handled in units called *fields*. Conceptually, a field is simply some number of consecutive bits at the same location in every processor. A field can be of any size. When a field is allocated, its size is specified by the user. Every field belongs to exactly one VP set. When we speak of allocating a field to a VP set, we mean allocating a field to each virtual processor in the VP set.

Most Paris instructions operate on memory fields. For example, a three-address **add** operation requires three field arguments: two source fields (whose contents are added together) and one destination field (into which the sum is stored).

There are two types of fields: heap fields and stack fields. Either type of field is allocated within a specific VP set. The distinction between heap and stack fields arises from the storage management strategy employed in physical CM memory in order to support virtual processors. Heap fields may be allocated and deallocated in any order. Stack fields may be allocated in any order, but they must be deallocated in the reverse order from that of their allocation. Thus, heap fields are more flexible, but have higher overhead than stack fields.

Processor Addresses

Paris supports two different sorts of addresses for virtual processors: the *send address*, which is used for general-purpose communication among virtual processors, and the *NEWS address*, which describes a VP's position in the *n*-dimensional grid used to optimize nearest-neighbor communication and scanning.

A virtual processor has one send address and one NEWS address at all times. Send addresses and NEWS addresses are specific to a VP set; that is, every VP in a VP set has a unique send address and a unique NEWS address, but it is possible for a VP in another VP set to have the same send address or NEWS address. Since Paris always operates within a single VP set (the current VP set), there is normally no ambiguity as to which VP is meant by a given address.

For communication from one VP set to another, Paris can uniquely identify the intended destination VP with ease. Communication across VP sets is effected with either the Paris **send** operation (and its variants) or the Paris **cross-vp-move** operation. Each of these operations expects the source field to be in the current VP set, while the destination field may be in the same or some other VP set. A field is always associated with exactly one VP

set, and this fact allows Paris to determine unambiguously the intended destination VP. The **send** operation sends data from each source VP to a destination that may be in the same VP set or in another VP set. The **cross-vp-move** operation transfers a copy of all or a portion of a field from one VP set to another according to a specified axis-mapping correspondence; it is faster than **send** but less general.

Flags

Each Paris virtual processor has an assortment of one-bit flags. Many Paris operations affect these flags rather than, or in addition to, storing results into the memory. For example, the **CM:s-add-2-1L** operation adds one signed integer to another, but also stores information into the carry flag and the overflow flag. Similarly, **CM:u-gt-2-1L** compares two unsigned integers and sets the test flag if the first is greater than the second.

The context flag is of particular importance. Most Paris instructions are *conditional*: a virtual processor participates only if its context flag is 1. A few instructions, including those that can alter the context flag itself, are unconditional.

9.2 Instruction Set Overview

This section provides a quick guided tour of some of the instructions in the Paris instruction set, organized by categories of functionally related operations. This is not a complete list of Paris operations. The names of the operations are presented in abbreviated form, glossing over the many addressing mode variants and, in some cases, data type variants. For example, the Paris **add** instruction comes in two-operand and three-operand forms, with or without an immediate operand, for signed integer, unsigned integer, floating-point, and complex floating-point data types; but for brevity only the generic operation name **add** is mentioned below. The full name of a Paris operation includes the prefix **CM_** (for Fortran or C) or **CM:** for Lisp, perhaps another prefix indicating data type, and suffixes indicating addressing modes. For example, "signed integer add immediate two-address" is **CM_s_add_constant_2_1L**.

For more information about Paris instructions, see *Connection Machine Parallel Instruction Set*, and *Connection Machine Programming in C/Paris*.

Memory Allocation

Paris provides operations that create, deallocate, and otherwise manipulate VP sets, geometries, and fields. For example, the operation **allocate-vp-set** creates a new VP set having a specified geometry. The geometry must first be defined using an instruction such as **create-geometry**. To create a field that may then be used as an argument to most other Paris operations, the instruction **allocate-stack-field** or **allocate-heap-field** may be used.

Arithmetic Operations

Paris provides most of the unary and binary arithmetic operations one might expect to find in a computer instruction set:

add	floor	eq	abs	subfrom
subtract	ceiling	ne	negate	divinto
multiply	truncate	gt	signum	add-carry
divide	round	ge	power	random
max	mod	lt	shift	
min	rem	le	scale	

A complete set of transcendental and trigonometric functions is also supported, including hyperbolic functions and their inverses:

exp	sin	asin	sinh	asinh
ln	cos	acos	cosh	acosh
sqrt	tan	atan	tanh	atanh

Compound operations are equivalent to sequences of simpler instructions but are specially coded for improved floating-point performance. The following compound Paris instructions are representative:

mult-add	add-mult
mult-sub	sub-mult

Operations on Bit Fields

Paris provides all ten nontrivial bitwise boolean operations:

```
logand          logeqv
logior          logandc1
logxor          logandc2
lognand         logorc1
lognor          logorc2
```

In addition, the **lognot** operation inverts all the bits of a field.

The following operations also treat fields in a bitwise fashion:

```
move            swap
move-reversed   latch-leds
```

The **move** operation copies one field to another; **move-reversed** reverses the order of the bits; **swap** exchanges the contents of two fields.

The red lights on the face of the CM-2 cabinet may be turned off and on under user program control with the **latch-leds** instruction; there is one light for every sixteen physical processors.

General Interprocessor Communication

The **send** and **get** operations use the router mechanism to transfer data to or from arbitrarily designated virtual processors, in either the current VP set or any other designated VP set.

The **send** operation takes a source field in the current VP set, a destination field that may be in any VP set, and an address field in the current VP set that contains addresses of virtual processors within the VP set of the destination field. Thus each virtual processor in the current VP set contains a source message and an indication of which virtual process is to receive it; the **send** operation transfers these messages all at once to their intended destinations.

It may happen that a given destination receives more than one message. The **send** operation has many variants, indicating how multiple received messages are to be handled:

```
send-with-overwrite        send-with-logand
send-with-logior           send-with-logxor
send-with-s-add            send-with-s-multiply
send-with-u-add            send-with-u-multiply
send-with-f-add            send-with-f-multiply
send-with-c-add            send-with-c-multiply
send-with-s-max            send-with-s-min
send-with-u-max            send-with-u-min
send-with-f-max            send-with-f-min
```

The **send-with-overwrite** version indicates that one (arbitrarily chosen) message should be retained and other incoming messages should be discarded. The others indicate a *combining operation* to be applied; for example, **send-with-f-add** indicates that incoming messages should be regarded as floating-point numbers, and the destination field should receive their sum. This set of standard combining operations is also used by many other Paris communications operations.

The **get** operation is inverse to **send**. If **send** is viewed as a "write" by each active processor into a global shared memory, then **get** is the corresponding "read" operation.

NEWS Communication

The operations **send-to-news** and **get-from-news** are analogous to **send** and **get**, described in the previous section. Instead of transferring data to or from an arbitrarily designated processor, they transfer data to or from a nearest neighbor within the current NEWS grid for the current VP set. Data may be transferred in either direction along any dimension of the grid. Within a two-dimensional grid, for example, the choices are north, east, south, or west (hence the acronym NEWS). Within a three-dimensional grid, there are six nearest neighbors. In any case, a processor in an n-dimensional grid has $2n$ nearest neighbors. The **send-from-news** and **get-from-news** operations are considerably more efficient than **send** and **get** for these particular patterns of data transfer. Because **send-to-news** causes all processors to transfer data in the same direction (all east, all north, or whatever), there is no possibility of message collisions, and so no combining variants are needed.

Scanning and Related Operations

Paris provides a number of powerful array-processing operations that perform both communication and computation in regular, grid-oriented patterns:

```
scan      global      reduce      spread      multispread
```

Each of these has variants corresponding to the same standard combining operations used by **send**. For example, the **scan-with-f-add** operation (also known as "parallel floating-point sum prefix") can compute all the partial sums along a particular NEWS axis; that is, every virtual processor receives the floating-point sum of the source values from all processors before it along the axis. This operation is quite fast, as it is supported by special hardware and is performed in parallel by special algorithms. The **global-f-add** operation is similar, but instead of computing all partial sums for each row, it computes a single sum of values from every virtual processor in the current VP set; the sum is returned to the front end. The **reduce-with-f-add** operation is the special case of **scan-with-f-add**; it computes only the sums of entire rows or colums, not all the partial sums. This special case is more efficient than a general **scan**. Similarly, **spread** is the special case of **scan** whereby the sum, or perhaps a designated element of the axis, is replicated so that all virtual processors along the axis receive a copy. Finally, the **multispread** operations optimize the case of spreading over more than one NEWS dimension at once. All these instructions can use any standard combining operation, so that one may compute partial products, largest or smallest values, or bitwise logical operations.

These specialized patterns of computation occur surprisingly often in data parallel applications. Paris provides them as primitive operations for programming convenience and computational efficiency.

Data Transfer between Processor Array and Front End

The operations **read-from-processor** and **write-to-processor** allow the front end to access any field within any virtual processor, one scalar item at a time.

The operations **read-news-array** and **write-news-array** transfer entire arrays or subarrays between an array on the front end and the NEWS grid of the current VP set. Their implementation is optimized for relatively high throughput.

Chapter 10

CM-2 Architecture

As explained in Chapter 1, a Connection Machine system consists of a parallel processing unit containing thousands of data processors, a front-end computer, and an I/O system that supports mass storage, graphic display devices, and VME and HIPPI peripherals.

The central element in the system is the parallel processing unit, which contains

- from 4K to 64K data processors
- a sequencer that controls the data processors
- an interprocessor communications network
- zero or more I/O controllers and/or framebuffer modules

See Figure 9. Each of the data processors can execute arithmetic and logical instructions, calculate memory addresses, and perform interprocessor communication or I/O. In this respect, each data processor is very much like an ordinary serial computer. The difference is that the data processors do not fetch instructions from their respective memories. Instead, they are collectively under the control of a single microcoded sequencer. The task of the sequencer is to decode commands from the front end and broadcast them to the data processors, which then all execute the same instruction simultaneously.

Interprocessor communication is particularly important in data parallel processing. Processors must be able to pass information among themselves in the pattern best suited to the needs of the moment; the patterns must be able to adapt to any application's needs, and to change over time. The CM-2 supports this need with three forms of communication within the parallel processing unit: routing, NEWS, and scanning.

This chapter describes the architecture of the parallel processing unit and its interprocessor communications network. Subsequent chapters describe the architecture of the I/O system and framebuffer.

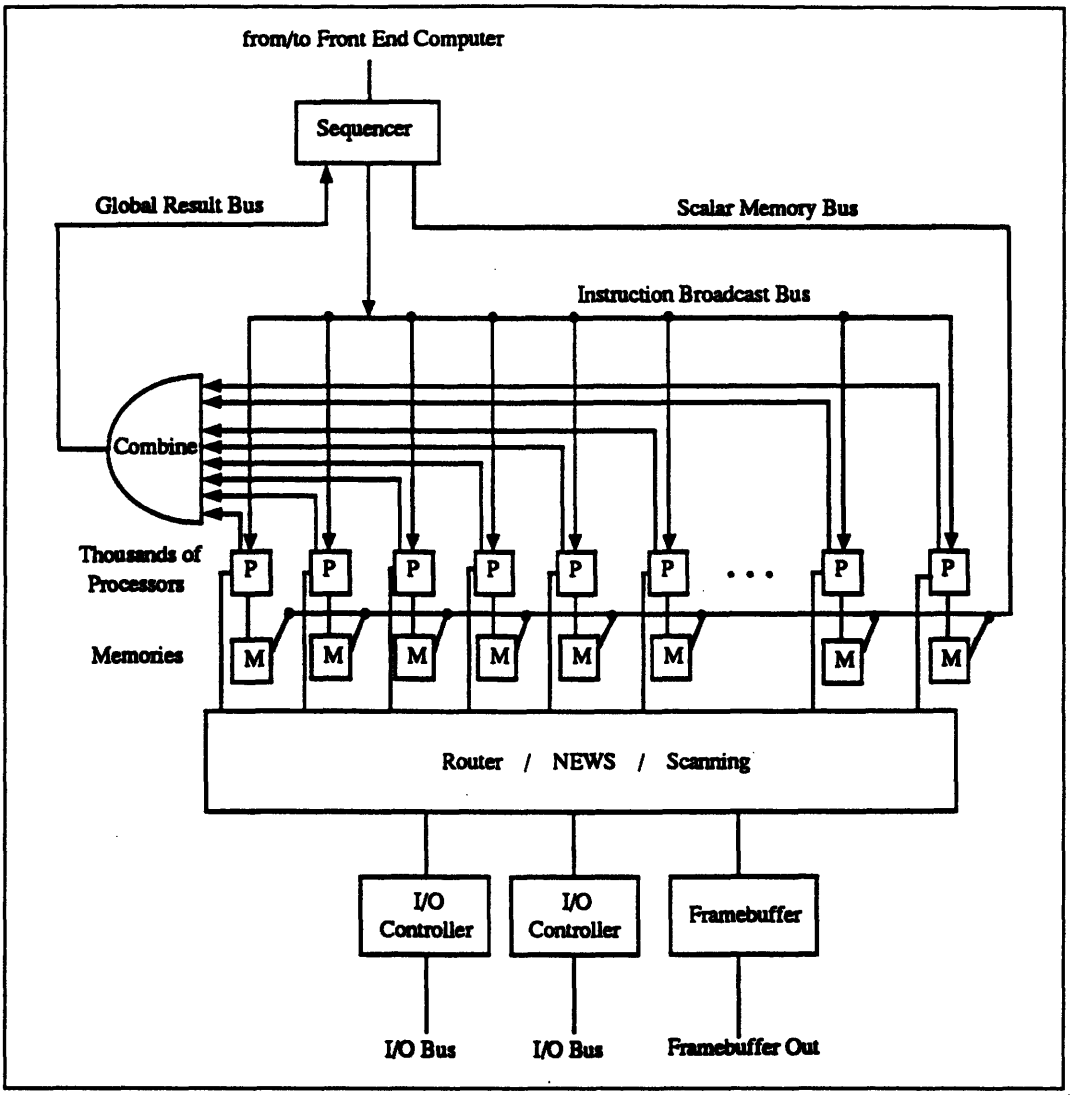


Figure 9. The CM-2 parallel processing unit

Parallel processing instructions issued by the front-end computer are received by the sequencer, which interprets them to produce a series of single-cycle "nanoinstructions." The nanoinstructions are broadcast over the instruction bus to thousands of data processors. Each data processor has its own memory.

All processors can access their respective memories simultaneously. Alternatively, the sequencer can access this memory serially, one 32-bit word at a time, over the scalar memory bus. The data processors can emit one datum apiece, and their combined value is delivered to the sequencer on the global result bus. The data processors can exchange information among themselves in parallel through routing, NEWS, and scanning mechanisms; these are in turn connected to the I/O interfaces.

10.1 Processor Architecture

Figure 10 diagrams the architecture of a CM-2 data processing node. As the figure shows, each node contains

- 32 CM-2 data processors, with their associated memory
- an optional floating-point accelerator
- communications interfaces for interprocessor communication

Each of the data processors houses an arithmetic-logic unit (ALU). Together, they create the CM-2's parallel processing array.

10.2 The Parallel Processing Array

A CM-2 arithmetic-logic unit consists of a 3-input, 2-output logic element and associated latches and memory interface.The basic ALU cycle first reads two data bits from memory and one data bit from a flag. The logic element then computes two result bits from the three input bits. Finally, one of the two results is stored back into memory and the other result into a flag. The entire operation is conditional on the value of a third flag, the context flag. If this flag is zero, then the results for its data processor are not stored.

The logic element can compute *any* two boolean functions on three inputs; these functions are simply specified as two 8-bit bytes representing the truth tables for the two functions. This simple ALU suffices to carry out, under control of the sequencer, all the operations of the Paris instruction set.

Consider, for example, addition of two k-bit signed integers. First the virtual processor context flag is loaded into a hardware flag register, to serve as the condition flag for all remaining ALU operations. Next a second hardware flag is cleared for use as a carry bit. Next come k iterations of an ALU cycle that reads one bit of each operand and the carry bit from memory, computes the sum (a three-way exclusive OR) and carry-out (a three-input majority function), and stores the sum back into memory and the carry-out back into the carry flag. These cycles start with the least significant bits of the operands and proceed toward the most significant bits. The last of the k cycles stores the carry-out into a different hardware flag, so that the last two carry-outs may be compared to determine whether overflow has occurred.

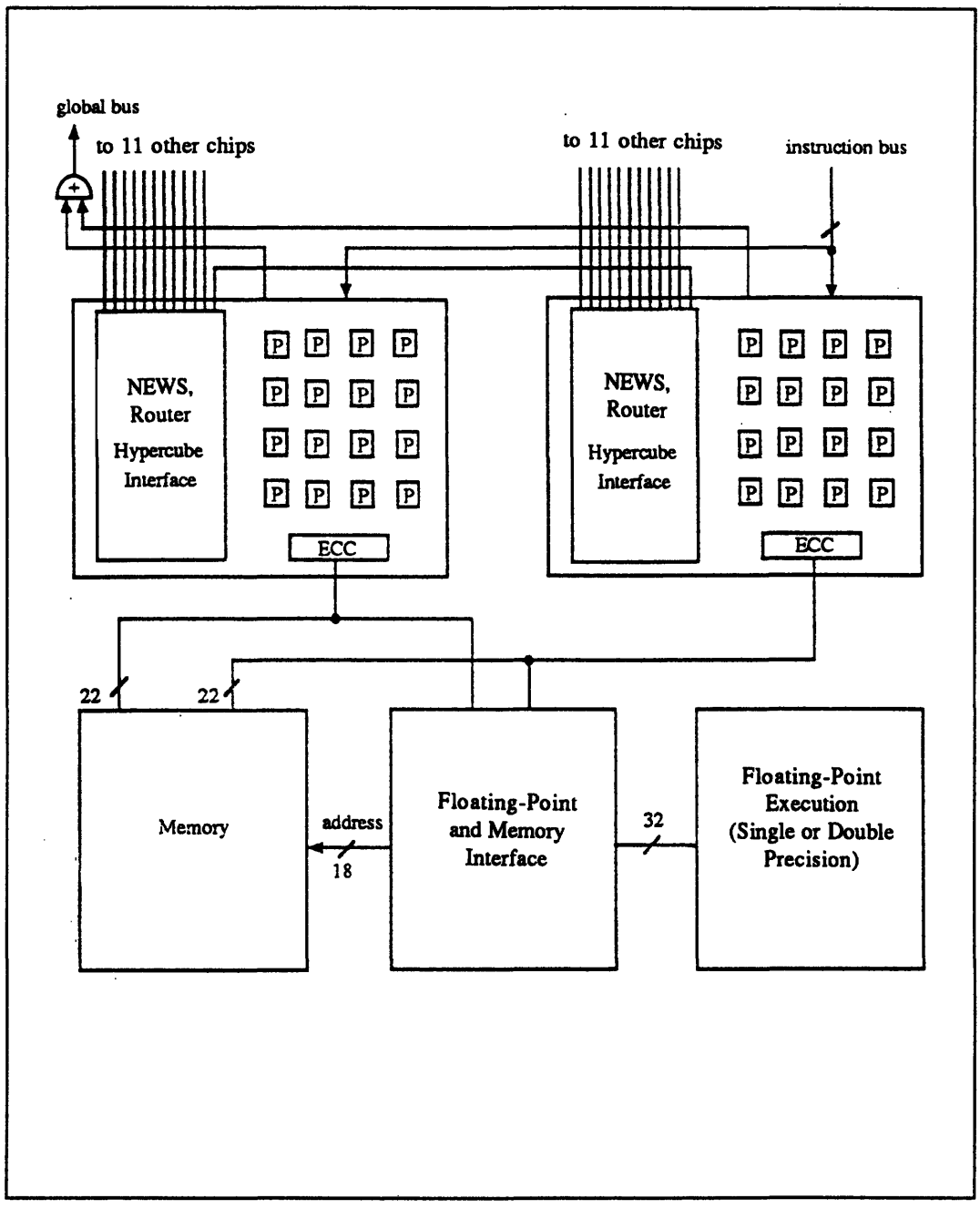


Figure 10. Two CM-2 processor chips with memory and floating-point chips

CM-2 processor chips are arranged in pairs. Each pair shares a group of memory chips, a floating-point interface chip, and a floating-point execution chip. The memory chips provide a 44-bit-wide data path; 22 bits (16 data and 6 ECC) go to each processor chip. The floating-point interface chip handles these same 44 bits and also provides memory address control for indirect addressing. The floating-point execution chip receives operands from the floating-point interface chip, 32 bits at a time, and in the same manner gives the interface chip results to be stored back into memory.

Integer arithmetic is thus carried out in a bit-serial fashion. At about half a microsecond per bit, plus a few more microseconds for instruction decoding and other overhead, a 32-bit add takes about 21 microseconds. If the ratio of virtual to physical processors (known as the VP ratio) *n* is greater than one, this process is repeated *n* times; one addition is performed for each virtual processor. With 64K processors all computing in parallel, this produces an aggregate rate of 2500 Mips (that is, 2.5 billion 32-bit integer additions per second). All other Paris operations are carried out in like fashion.

The ALU cycle is broken down into subcycles. On each cycle the data processors can execute one low-level instruction (called a nanoinstruction) from the sequencer and the memories can perform one read or write operation. The basic ALU cycle for a two-operand integer add consists of three nanoinstructions:

```
LOADA:   read memory operand A, read flag operand, latch one truth table
LOADB:   read memory operand B, read condition flag, latch other truth table
STORE:   store memory operand A, store result flag
```

Other nanoinstructions direct the router, NEWS grid, hypercube interface, and floating-point accelerator; initiate I/O operations; and perform diagnostic functions.

10.3 The Floating-Point Accelerator

In addition to the bit-serial data processors described above, the CM-2 parallel processing unit is typically configured with a floating-point accelerator that is closely integrated with the processing unit. The floating-point accelerator operates in both single precision (32 bits) and double precision (64 bits) and supports IEEE standard floating-point formats and operations.

As Figure 10 shows, floating-point acceleration hardware consists of two special-purpose VLSI chips for each pair of CM-2 processor chips: a memory interface unit and a floating-point execution unit.

As an example of the operation of the floating-point accelerator, consider the execution of a two-operand floating-point instruction such as **`add`** or **`multiply`**. Execution proceeds in five stages; each stage consists of 32 nanoinstruction cycles (one cycle for each of the 32 data processors on the two CM-2 processor chips).

1. The first operand for each of 32 data processors is transferred from memory to the interface chip.

2. The first operand is transferred from the interface chip to the floating-point execution chip. (The floating-point execution chip is capable of storing 32 values of a given precision.) Simultaneously, the second operand is transferred from memory to the interface chip.

3. The second operand is transferred from the floating-point interface chip to the floating-point execution chip, where the operation is performed. At the end of this stage, the floating-point execution chip contains the 32 results.

4. The results are transferred from the floating-point execution chip to the interface chip.

5. The results are transferred from the interface chip to memory.

If the VP ratio is n, this process is pipelined so as to require only $3n+2$ stages instead of $5n$ stages. Thus one floating-point operation is performed for every three memory references (the same performance as for integer operations). More specialized instructions, such as **`scan`** or **`global`** reductions, or convolution and FFT, achieve even greater operation rates by performing several operations and/or accumulating results on the floating-point execution chip.

Just as 64-bit integer operations take twice as long as 32-bit integer operations, so double-precision floating-point operations (using the double-precision floating-point accelerator) take twice as long as single precision in the simple cases because memory bandwidth is the limiting factor. More complex operations may take less than twice as long in double precision because of their improved ratio of floating-point operations to memory references.

10.4 The Router

Interprocessor communication is accomplished in the CM-2 parallel processing unit by special-purpose hardware. Message passing happens in parallel; all processors can simultaneously send data into the local memories of other processors, or fetch data from the local memories of other processors into their own. The hardware supports certain message-combining operations: that is, the communication circuitry can combine multiple messages en route to the same destination processor by applying some arithmetic or logical combining operation. The destination processor receives the combined result.

The most general of the CM-2's communications mechanisms is the router, which allows any processor to communicate with any other processor. One may think of the router as allowing every processor to send a message to any other processor, with all messages being

sent and delivered at the same time. Alternatively, one may think of the router (in conjunction with indirect addressing) as allowing every processor to access any memory location within the parallel processing unit, with all processors making memory accesses at the same time.

Each CM-2 processor chip contains one router node, which serves the 16 data processors on the chip. See Figure 10. (Note that because a processing node contains two processor chips, each processing node contains *two* router nodes.) The router nodes on all the processor chips are wired together to form the complete router network. The topology of this network is a boolean n-cube. For a fully configured CM-2 system, the network is a 12-cube connecting 4,096 processor chips. Each router node is connected to 12 other router nodes; specifically, router node i (serving data processors $16i$ through $16i+15$) is connected to router node j if and only if $|i-j| = 2^k = (j \oplus k) \vee 2^k$ for some integer k (that is, i and j differ only in bit position k), in which case we say that routers i and j are connected along dimension k.

Each message travels from one router node to another until it reaches the chip containing the destination processor. The router nodes automatically forward messages and perform some dynamic load balancing. For example, suppose that processor 117 (which is processor 5 on router node 7, because 117 = 16 x 7 + 5) has a message M whose destination is processor 361 (which is processor 9 on router node 22). Since $22 = 7 + 2^4 - 2^0$, this message must traverse dimensions 0 and 4 to reach its destination. In the absence of contention for hypercube wires, router 7 forwards the message to router 6 ($6 = 7 - 2^0$), which forwards it to router 22 ($22 = 6 + 2^4$), which delivers the message to processor 361. On the other hand, if router 7 has another message that needs to use dimension 0, it may choose to send message M along dimension 4 first, to router 23 ($23 = 7 + 2^4$), which then forwards the message to router 22, which then delivers it.

The algorithm used by the router can be broken into stages called *petit cycles*. Each petit cycle consists of enough ALU/route cycles to process all the bits of a destination address and a message. (An ALU/route cycle is a **LOADA**, **LOADB**, **STORE** ALU cycle, followed by a **ROUTE** nanoinstruction.) The delivery of all the messages for a Paris **send** operation might require only one petit cycle if only a few processors are active, but if every processor is active then typically many petit cycles are required. It is possible for a message to traverse many dimensions, possibly all twelve, in a single petit cycle, provided that contention does not cause it to be blocked; the message data is forwarded through multiple router nodes in a pipelined fashion. A message that cannot be delivered by the end of a petit cycle is buffered in whatever router node it happens to have reached, and continues its journey during the next petit cycle. If petit cycles are regarded as atomic operations, then the router may be viewed as a store-and-forward packet-switched network. Within a petit cycle, however, the router is better regarded as a circuit-switched network, where dimension wires are

assigned to particular messages whose contents are then pumped through the reserved circuits.

Each router node has a limited ALU, distinct from those for the data processors. During each petit cycle, each router node checks to see if its buffers hold several messages that are all going to the same processor. If so, the messages are combined. This may be done by taking the numerically greatest, summing them, taking the bitwise logical OR, or arbitrarily discarding all but one message. Other combining functions are implemented in terms of these primitives. For example, combining with bitwise logical AND is performed by inverting the original message data, sending it with OR-combining, and re-inverting received messages. (Such tricks are implemented by the sequencer and are transparent to the Paris user.) This hardware support for combining accelerates Paris instructions that perform combining operations. The combining hardware also combines **`read`** requests during execution of the Paris **`get`** instruction, so that a value fetched once from a processor can be returned to many requestors.

Each router node also contains specialized logic to support virtual processors. When a message is to be delivered by a router node, it is placed in the correct region of memory for the virtual processor originally specified as the message's destination.

10.5 The NEWS Grid

Communication operations between processors that are nearest neighbors within a Cartesian grid are much more efficient than the general router mechanism because they exploit three different transfer methods, two of which have special hardware support.

Imagine a VP set with 2^{22} virtual processors (about four million of them) running on a 64K Connection Machine system. We wish to organize these virtual processors as a 2048 x 2048 grid. (Such grids are referred to as NEWS grids because each processor has a north, east, west, and south neighbor.)

The system has 2^{12} processor chips with connecting wires forming a boolean 12-cube; these are the same physical wires that serve the general router mechanism. A subset of these wires can be chosen so that they connect the 2^{12} chips as a two-dimensional grid of shape, say, 64 x 64. (This uses the technique of Gray-coded grid coordinates, a method of assigning a series of coordinates so that adjacent coordinates differ in exactly one bit position —which implies that the processors they label will have a hypercube wire between them. There are many ways to choose the subset of connecting wires, and many other shapes are also possible, but for this example we will assume a 64 x 64 configuration.)

Within each chip are 16 physical processors. Imagine that they are arranged within the chip as a 4 x 4 grid. Within each physical processor are 64 virtual processors. Imagine that they are arranged within the chip as an 8 x 8 grid.

Suppose we wish each processor to send a value to its neighbor to the east. Within each group of 64 virtual processors, 56 of them must send data to another virtual processor that is within the same physical processor. This is the first of the specialized transfer methods: seven-eighths of the work is done by having each physical processor rearrange data within its own memory, without doing any physical interprocessor communication.

The rest of the work requires each physical processor to send eight messages to its physical processor neighbor to the east. Within each group of 16 physical processors, 12 of them must send data to another physical processor that is within the same chip. This is the second of the specialized transfer methods: three-fourths of the remaining work is done by a specialized per-chip permutation circuit that is independent of the entire router/hypercube-wire mechanism.

The last bit of work requires each chip to send 32 messages to its neighbor along one hypercube wire (and to receive 32 other messages from its neighbor to the west, along some other hypercube wire). This is the third of the specialized transfer methods: because the communication pattern is so regular, the per-chip permutation circuit can determine directly the wires to be used, thus avoiding the need to calculate the address of a neighbor and feed the address to the router, bit by bit.

The hardware is flexible enough to accommodate any shape or VP ratio. For example, the per-chip permutation circuit can organize its 16 physical processors as 8 x 2, or 1 x 16, or 4 x 2 x 2, or 1 x 8 x 2, or 2 x 2 x 2 x 2, and so on. Thanks to this specialized hardware support, NEWS grids of any shape or number of dimensions can be handled with great speed and efficiency.

10.6 Scans and Spreads

Scanning is a powerful operation on NEWS grids that combines communication and computation. Simultaneously, in every row of a grid along a particular dimension, scanning computes all the partial sums of that row. This is an extraordinarily powerful operation. Instead of sums one may compute products, find the largest or smallest value, or compute bitwise AND, OR, or exclusive OR. Special cases of scanning include finding the sum (product, largest value, etc.) over all elements of an array, finding the sums of all rows or columns of a matrix, and replicating a vector so as to fill a matrix.

Spreads, in contrast, allow a value from one processor to be sent to all other processors. This requires transmitting the value from one chip to all other chips. By using the 12 wires on each chip that form the CM-2's 12-cube, a single-bit value can be spread from one chip to all other chips in only 75 steps. Furthermore, if the processors are programmed to perform bit-serial arithmetic as the bits fly past, the same communication pattern yields partial sums or other scan operations.

Variants of these techniques encompass all the specialized scan, reduce, spread, and multi-spread operations on grids of any dimension, and are encapsulated as Paris instructions for ease of access.

10.7 Communication with the Front End

All Connection Machine programs execute on a front end, with the front end issuing instructions as needed to the CM-2 parallel processing unit. The sequencer then breaks down these "macroinstructions" into appropriate "nanoinstructions" and broadcasts these instructions to all the processors at once.

In contrast, data can be exchanged between the front end and the processing array in any one of three ways: *broadcasting*, *global combining*, and the *scalar memory bus*. Broadcasting allows a single value from the front end to be replicated and sent to all the data processors at once. Global combining allows the front end to obtain the sum, largest value, logical OR, or whatever, of one value from each data processor. The scalar memory bus allows the front end to read or write one 32-bit value at a time anywhere in any processor.

Chapter 11

Data and Image I/O

As with most computer systems, the Connection Machine's I/O operations can directly affect the system's overall efficiency and ease of use. For Connection Machine applications in particular, where large data files are routine, high I/O bandwidth can significantly improve job execution efficiency as well as program development times. In addition, the complex problems that the Connection Machine is routinely used to investigate, and the massive data sets they usually involve, impose difficult conceptualization and interpretation demands on the programmer/scientist. Such users can benefit greatly from graphic representation of their computations, displayed in high-resolution color with real-time animation.

The Connection Machine supports these ancillary requirements with high-performance data I/O and imaging systems. This chapter provides an overview of these facilities. Chapters 12 through 14 contain more detailed descriptions of the individual I/O and visualization system components.

11.1 Data I/O Channels

Every Connection Machine has, depending on its size, from 2 to 16 channels available for data and/or image I/O. These channels are organized into pairs, with each pair controlled by a separate section of the CM. For example, an 8K CM-2 has 2 I/O channels, as does a 4K CM-2a. At the other end of the spectrum, a 64K CM-2 has 16 I/O channels, two for each of its eight sections.

The Connection Machine supports simultaneous use of multiple I/O channels with the restriction that only one channel in any given pair can be active at time. This means that an 8K CM-2 is limited to using one I/O channel at a time, while a 64K CM-2 can have up to 8 of its 16 channels in use simultaneously.

A channel's mode of use is determined by the type of interface board that is installed in its backplane slot — a Connection Machine I/O controller (CMIOC) or a framebuffer.

- A CMIOC board adapts its channel for transferring data to and from external devices over a dedicated data bus, called the CMIO bus.
- A framebuffer board adapts its channel for connection to a graphics display monitor.

Although there are no hardware or software restrictions against filling both I/O slots on the same backplane with the same type of I/O module (either two CMIOCs or two framebuffers), most configurations have a single CMIOC and/or a single framebuffer serving each section.

11.2 Data I/O Overview

From the user's perspective, the CM's data I/O system is organized as a set of client and server processes running on two or more computers that are linked by Ethernet. Included in this network will be at least one Connection Machine front end running a user application program. Other computers in the net are located in I/O devices that are connected to the Connection Machine via one or more CMIO buses. These I/O computers run file server processes for their respective devices.

A user program becomes a client when it initiates a CMIO operation. File system software on the front end issues commands to the file server on the appropriate I/O computer over the Ethernet link. The file server, in turn, initializes the data storage or I/O processor device on which it resides for the impending transfer. For a Connection Machine write to a data storage device, for example, the file server translates the logical file description it receives from the front end into a physical description meaningful to the device, including the size of the transfer in device-specific units and the physical location at which the file is to be stored. The file server also passes device status information back to the front end.

The front end also sends I/O instructions to the CM. There, these instructions invoke microcoded I/O service routines, which control the CM's half of the I/O operation. For a Connection Machine write operation, for example, this control involves initializing the CMIOC for the transfer and moving data from the parallel processing unit to data buffers on the CMIOC.

A CMIOC module serves as the bridge between a set of physical processors and a CMIO bus. The CMIO bus, in turn, links the CM to a world of peripheral devices, including:

- DataVault: a disk-based mass storage system (see Chapter 12)
- CM-HIPPI system: a CMIO-to-HIPPI bus interface (see Chapter 13)
- CM-IOP system: a CMIO-to-SCSI bus interface (see Chapter 13)
- VMEbus interface controller — a third-party computer equipped with a VMEIO board to provide a CMIO-to-VMEbus interface (see Chapter 13)

The DataVault, CM-HIPPI, CM-IOP, and VMEIO board are all manufactured by Thinking Machines Corporation.

Each CMIOC and CMIO bus can support peak data transfer rates as high as 50 megabytes per second. I/O rates exceeding 100 megabytes per second can be achieved using multiple CMIOC boards with a separate CMIO bus connected to each and spreading files across all the buses. Not all I/O devices can maintain this rate, however.

Some consideration should be given to how the CMIO bus configuration can be arranged to match the device I/O bandwidth to the Connection Machine's I/O capability. Section 11.5, CMIO Bus, touches briefly on this subject.

11.3 Graphics Output for Data Visualization

The Connection Machine system includes a powerful set of visualization tools that support display of memory contents via a direct link to a high-resolution color monitor or to any X windows server connected to the front end. The framebuffer module, which is installed in one of the Connection Machine backplane slots reserved for I/O modules, provides the direct link between the Connection Machine parallel processing unit and a 19-inch color monitor. The Connection Machine visualization hardware is described in Chapter 14.

11.4 CM I/O Controller

Each CMIOC serves the processor boards that are plugged into the same backplane as itself, treating the 8K physical processors on its backplane as two banks of 4K processors. Each 4K bank is distributed evenly among 256 processor chips (16 processors per chip) and each chip is connected to the backplane by a single I/O line. A bank can therefore pass 256 bits in parallel to its associated CMIOC. Bytewise parity is generated for the data sent to the CMIOC; data received from the CMIO bus also carries bytewise parity. This contributes an

additional 32 bits to the I/O path, yielding a 288-bit-wide path between the processors and the CMIOC.

Only one bank on a given backplane can be sending or receiving data at a time. During I/O operations, the sequencer controls a bank switch that determines which bank in its 8K set is active. It toggles between the banks on alternate bits.

A CMIOC buffers its I/O data in a FIFO that is 288 bits wide by 512 bits deep. The CMIOC checks the parity it receives from the processor boards with outgoing data (processor-to-CMIO bus) and generates new bytewise parity for the data it drives onto the CMIO bus. Incoming data has its parity checked first on the CMIOC and then again on the processor boards.

The data path on the CMIO bus is 72 bits wide, consisting of 64 data bits and 8 parity bits. Consequently, the CMIOC must also multiplex and demultiplex data passing between the 288-bit-wide processor data path and the 72-bit-wide CMIO bus.

11.5 CMIO Bus

The CMIO bus is based on a multi-drop architecture; up to 16 devices, including the CMIOC, can be connected to a single bus. As mentioned earlier, the Connection Machine can have up to 16 CMIOCs, each of which can be connected to a separate CMIO bus. Thus, a fully configured 64K CM-2 could physically accommodate 240 external I/O devices, plus the 16 CMIOCs needed as interfaces to the 16 buses.

The combination of multiple CMIOCs and multi-drop bus architecture offers valuable flexibility in designing an optimal CM I/O configuration. One goal in designing CM I/O system configuration is achieving a good match between the bandwidth of the external devices and the Connection Machine's own I/O capability.

For example, most peripheral devices do not operate at the speeds the Connection Machine is able to sustain. It can be advantageous, then, to connect multiple devices to the Connection Machine in parallel; that is, to connect each device to a separate CMIO bus, each of which is connected to a separate CMIOC. This arrangement allows data transfers to be conducted in parallel, yielding aggregate transfer rates well above the rate possible with a single peripheral transferring 64-bit-wide data. This approach is called file striping.

File striping on the CM uses two, four, or eight CMIO buses connected in parallel to separate CMIOCs, with a separate DataVault on each bus, producing an arrangement called a striped set. The file system distributes its files in 512-bit slices across all the processors in the striped set. When a file is written to the DataVaults, it is sent in parallel to all the

DataVaults in the striped set. Each DataVault receives a 512-bit chunk of the file in 64-bit consecutive units. Adjacent DataVaults receive contiguous 512-bit chunks of the file. Files are retrieved from the striped set in the reverse order, with the file system software distributing the 512-bit chunks to the appropriate 8K sets of processors.

Alternatively, one or more peripherals may be connected to a single bus. The bus, in turn, is connected in daisy chain fashion to CMIOCs in more than one section of the parallel processing unit. This configuration allows data to be moved directly between any device on the bus and any part of the processing unit.

Most CM I/O systems employ a mix of these configurations. Some devices are connected to multiple CMIOCs, while others connect to just one. The Connection Machine router is used as necessary to move data to its intended destination in the parallel processing unit.

In addition to its 64 data lines and 8 parity lines, the CMIO bus includes 8 control lines. These are used to implement an asynchronous transfer protocol. This protocol requires one of the devices on the bus to serve as bus arbiter. Any device wishing to transfer data to another device on the bus submits a request to the arbiter to be given control of the bus. The arbiter reconciles conflicting bus requests and allocates bus mastership, following a scheme designed to avoid bus monopoly by any one device.

Each device's interface to the CMIO bus is implemented by a state machine. When a device participates in a transfer across the bus, its state machine controls one end of the transfer handshaking. The bus interface also recognizes and flags bus protocol errors that may occur during the transfer. This error detection allows the I/O control software to retry a transfer a prescribed number of times.

Chapter 12

The DataVault

The DataVault is a disk-based mass storage system for Connection Machine files. It combines large capacity with high transfer rates and exceptional reliability, making it well suited for storing and transferring massive data files.

The basic DataVault disk configuration provides storage for 30 gigabytes of data; this can be expanded to 60 gigabytes. A DataVault is capable of transferring data at a sustained rate above 25 megabytes per second.

For further efficiency, the CM file system allows files to be striped across a set of two, four, or eight DataVaults. Each DataVault in the set is attached, via a separate CMIO bus, to a separate CMIOC; but the whole set acts as a single logical device, under the control of one of its member servers. A set of striped DataVaults thus writes and reads in parallel to and from two, four, or eight sections of the CM-2. A set of eight DataVaults, attached to a 64K CM-2, can hold files of up to 480 gigabytes (8 x 60 gigabytes) and achieve data transfer rates well above 100 megabytes per second.

A 30-gigabyte DataVault employs an array of 42 $5^1/_4$-inch Winchester disk drives, of which 39 are active and 3 are spares. (See Figure 11.) Of the 39 active drives, 32 hold data and 7 hold error correction code (ECC) bits. The ECC bits allow the DataVault to correct single-bit errors and to flag multiple-bit errors in each 32-bit value retrieved from the disks. A 40-gigabyte DataVault has 84 drives, of which 64 hold data, 14 hold ECC bits, and 6 are spares.

The data is spread across the drives, one bit per drive. Each 64-bit data chunk received from the Connection Machine I/O bus is split into two 32-bit words. After verifying parity from the I/O bus, the DataVault controller adds 7 ECC bits and stores the resulting 39 bits on 39 individual drives. Subsequent failure of any one of the 39 drives does not impair reading of the data, since the ECC data allows any single-bit error to be detected *and* corrected for every data word. The ECC data permits 100% recovery of the contents of a failed disk, allowing a new copy of this data to be reconstructed and written onto a spare disk. Once this recovery is complete, the data base is healed.

The DataVault's speed and large capacity support the Connection Machine in a number of ways. Loading Connection Machine memory with data for processing is a prime example. Most source media for Connection Machine data have slow data transfer characteristics — typical transfer rates for tape drives on a VMEbus, for example, are on the order of 5 to 10 megabytes per second. The rate at which Connection Machine memory is loaded can be

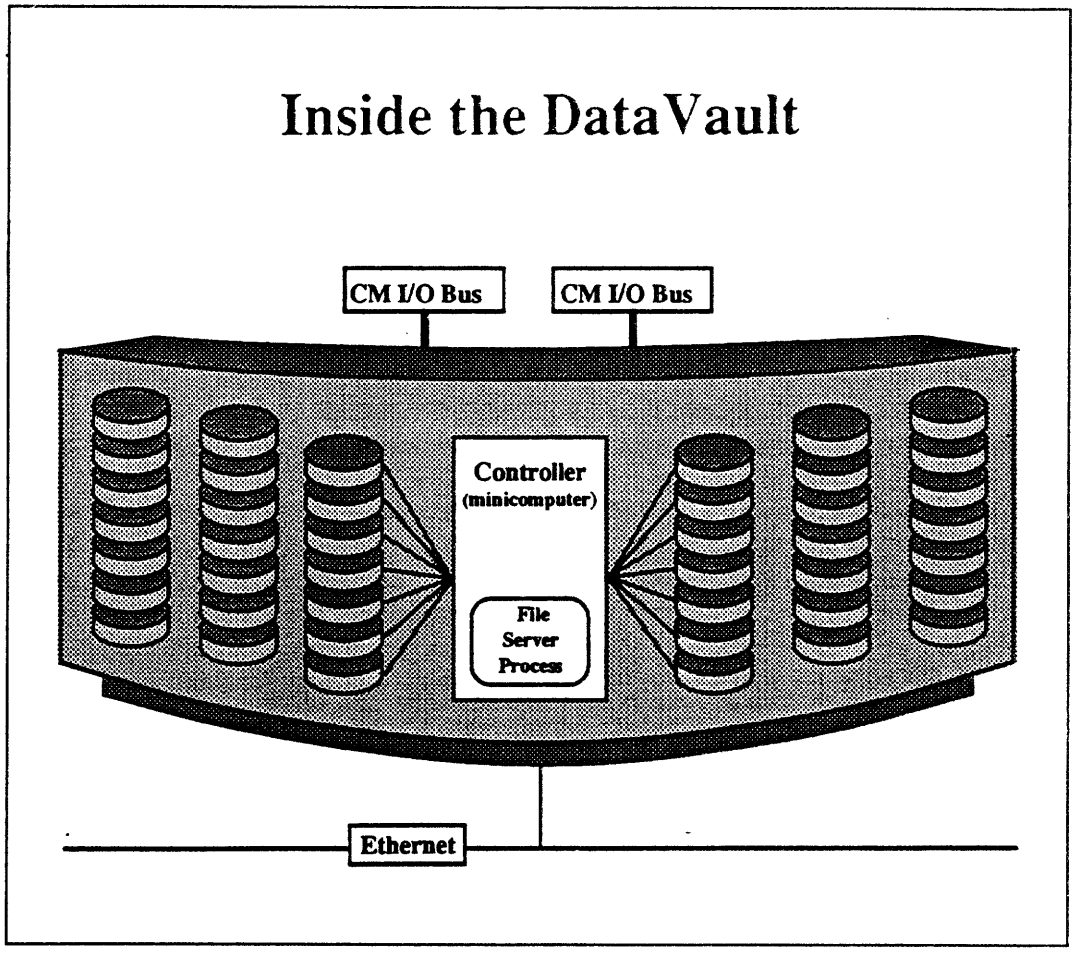


Figure 11. Inside the DataVault

The DataVault cabinet contains 42 disk drives (or 84 for double capacity) plus a standard minicomputer to control them. The controller accepts I/O commands over an Ethernet connection and transfers data over a high-speed CMIO bus.

For every 32-bit data word written to the DataVault, each bit goes to a different disk drive. Seven error correction code (ECC) bits generated by the DataVault on the data it receives go to seven more drives. The remaining three drives serve as spares. Even if a drive should fail totally, its contents can be reconstructed and written to one of the spare drives. The ability to recover from such component failures makes the DataVault storage system remarkably reliable.

greatly improved over this by using the DataVault as a way station for the data. The data is written first to the DataVault at the tape drive's transfer rate and then loaded from the DataVault to Connection Machine memory at the DataVault's rate.

The tape-to-DataVault path would include either a VMEIO or SCSI bus interface controller, depending on the type of I/O interface used by the tape unit. This bus interface would provide a path between the tape device and the CMIO bus. It would also perform CMFS functions for the tape unit (see Chapter 13). This path could also be used when writing processed Connection Machine data out to tape files for long-term storage. The data would first be written out to the DataVault and then transferred to the tape.

The Connection Machine need not participate in the tape-to-DataVault transfers, leaving it free to perform other operations while these transfers are going on. Checkpointing and time sharing on the Connection Machine also benefit substantially from the DataVault's high bandwidth.

12.1 The File Server

All DataVault operations take place under the control of a standard minicomputer (built into the DataVault) running a file server process. The server manages the Connection Machine file system's UNIX-like hierarchical directory structure, handling the allocation of physical disk space and matching file names and logical read/write requests to the physical locations of data on the DataVault disks.

The file server receives commands from the front end via its Ethernet link. File server commands include creating files, as well as opening, reading, writing, and checking status. Operations that do not involve data transfers, such as opening or closing a file or sending status information to the front end, the file server performs itself, without interacting with the rest of the DataVault. For read/write operations, the file server issues the appropriate commands to the DataVault's internal controller.

The file server maintains duplicate copies of its file bit maps on two independent file server disks. The bit maps are used to map logical Connection Machine file system references to the corresponding physical disk blocks. This redundancy ensures continued access to user files on the DataVault even if one of the file server disks fails. The DataVault file server uses a form of memory caching to promote faster access of files. Whenever the file server opens a user file, it copies the corresponding physical block location information to its main memory. This allows the file server to produce physical addresses immediately during subsequent read and write operations.

Internally, the file server represents a Connection Machine file as a series of *extents*, or areas of contiguous disk surface. Each extent starts at a logical offset within the Connection Machine file; each has a physical disk address and a length. This representation allows a file to have arbitrarily large physically contiguous blocks of the disks holding data for logically contiguous segments of the file. As a result, positioning of the read/write heads is more efficient, yielding faster file access.

12.2 Writing and Reading Data

Data transfers move information between parallel variables in Connection Machine memory and DataVault files. A single read or write moves a specified number of bits into or out of each Connection Machine virtual processor. The principal events involved in writing a file to the DataVault are summarized below. Reading a file from the DataVault into Connection Machine memory is very similar and can be inferred from the write operation description.

A write operation is initiated by the front end. The front end issues a write instruction over the Ethernet to the DataVault file server. When it receives the logical file request, the file server translates the request into a series of physical disk addresses. Assuming that the request parameters are valid (e.g., there is sufficient space), the file server returns a message to the front end indicating successful completion. If the request cannot be fulfilled, the file server sends a failure report to the front end instead.

Data from Connection Machine memory is moved to the CMIOC, with parity checked for each byte, and stored in the buffer memories (288 bits wide by 512 bits deep) on those controllers. When the buffers are sufficiently full, the I/O controller signals its readiness to send data to the DataVault.

Data in the CMIOC is split into 72-bit units (64 data bits plus 8 parity bits). These 72-bit units are multiplexed out to the DataVault via the CMIO bus.

The DataVault checks and then strips the parity from the data it takes off the CMIO bus. Its ECC circuits then generate 7 bits of ECC for each 32 bits of the original data. The resulting 39 bits are distributed to 39 separate disk buffers, one bit per buffer. As these buffers fill up, the data is written out to the individual disks.

When all data has been written on the disks, a signal is returned to the front end that the transfer is complete.

Data being read into the Connection Machine memory from the DataVault follows the same path as for writing, but in reverse order, through the disk buffers, the CMIO bus, and the CMIOC. The data coming off the disks is checked by ECC circuits. Single-bit errors are corrected and logged, and the data is written with parity to the CMIO bus.

12.3 Data Protection

A transfer status may indicate that a single disk drive is failing and that the ECC was required to correct data. This will most often be discovered when the error logs are checked (typically at the end of the day). At that point, the faulty drive can be physically replaced with an external spare. If the site does not currently have any spares available in storage, other than the three (or six) spare drives contained in the DataVault, one of these internal spares can be logically substituted for the failing drive.

This logical substitution uses a software procedure, called *sparing*, that reconstructs the corrupted data, using the ECC circuits to correct the failing bit, and stores it on one of the spare drives provided for the purpose. The sparing program redirects the path followed by the faulty bit from the failing drive to the spare. Regeneration of this data takes from 30 to 60 minutes (two minutes per gigabyte), after which the data is again protected against the failure of another drive.

When the failed drive is physically replaced, the files are reconstructed using the same technique as is used when sparing the failed drive.

Chapter 13

CMIO Intelligent Bus Interfaces

The Connection Machine I/O system includes three products that are designed to link the Connection Machine to a heterogeneous world of data acquisition and storage. These products provide interfaces between the CMIO bus and I/O devices that are attached to HIPPI, SCSI, or VME buses.

13.1 HIPPI Bus Interface

The CM-HIPPI is a bus interface controller that is designed to transfer data at high speed between the ANSI draft standard HIPPI bus and one or more CMIO buses. It is primarily intended to link the Connection Machine and its DataVault to other supercomputer systems via two simplex HIPPI buses, one carrying incoming data and the other carrying outgoing data. Each HIPPI bus has a bandwidth of 100 megabytes per second.

The CM-HIPPI is a complete, integrated system. It contains a Sun-4/300 CPU, two disk drives, a VMEbus, HIPPI input and output interface modules, and up to eight HIPPI-to-CMIO interface modules. This architecture supports full duplex communication between a 32-bit HIPPI source/destination and multiple CMIO buses at a peak bandwidth of 200 megabytes per second.

The HIPPI controller CPU receives CMFS commands from the Connection Machine front end over an Ethernet cable. A file server process running on the CPU interprets these commands and controls the I/O operations engaged in by the CM-HIPPI accordingly. The disk drives store duplicates of the system software, the file server, and hardware diagnostic programs.

Together, the HIPPI input and output modules provide a full duplex I/O interface between a pair of external HIPPI buses and a pair of internal buses, one for incoming data and one

for outgoing data. These internal buses are also connected to the eight HIPPI-to-CMIO interface modules via a set of multiplexing switches. These switches provide the means for establishing and breaking links between specific CMIO buses and the HIPPI input and output ports.

Each HIPPI-to-CMIO module provides a separate path between a CMIO bus and the internal HIPPI buses, as controlled by the switch matrix. In this way, up to eight CMIO buses can be connected to the HIPPI input and output ports in parallel. Depending on the transfer rates of the various CMIO bus devices involved, the peak aggregate I/O potential of this configuration is 200 megabytes per second.

13.2 VMEbus Interface

The CMIO-to-VMEbus interface is based on Thinking Machines Corporation's VMEIO board. This is a printed circuit module that can be installed in a VME computer running UNIX or a UNIX-equivalent operating system. The combination of the VMEIO module and a UNIX-based computer provides an intelligent link allowing any device on the CMIO bus, including the Connection Machine and the DataVault, to transfer files to and from various I/O devices such as tape drives and video frame grabbers that have VMEIO buses.

The computer serves as a platform for a file server process, which interprets CMFS file operation commands. The Connection Machine's front end sends these commands to the file server and receives status messages from it via an Ethernet link. This computer also runs a special device driver for controlling the VME I/O functions.

The VME computer may also contain client processes that issue commands to servers in other computers on the CMIO bus, such as the DataVault's computer. Such processes may transfer data across the CMIO bus without requiring participation by the Connection Machine or its front-end computer. This is the arrangement described in Chapter 12.

The VMEIO module provides the hardware interface between a standard 32-bit VMEbus and the 64-bit (plus parity and control) CMIO bus. Transfer-rate differences between the two buses are alleviated by an 8-megabyte buffer (64 bits wide, 1 megabyte deep).

13.3 SCSI Bus Interface

The VMEIO module is a key component in the CM-IOP, a bus interface controller that provides a bridge between the CMIO and the industry-standard SCSI buses. The CM-IOP is a fully integrated system that includes a Sun-4/300 CPU, two disk drives, a VMEbus, a VMEIO module, and up to eight VME-to-SCSI interface control modules.

The Sun-4/300 runs a file server process that interprets CMFS commands received from the front end via an Ethernet connection. The two disk drives contain redundant copies of the CPU's system software as well as the file server and hardware diagnostic programs. The CPU communicates with the other circuit modules in the CM-IOP via an internal VMEbus.

The VMEIO module is the interface between the internal VMEbus and the system CMIO bus, enabling devices on the CMIO bus to transfer data into and out of the CM-IOP.

The SCSI interface controllers, in turn, connect the internal VMEbus to external SCSI-based devices, such as Storage Tek 4980 tape drives. The CM-IOP logic chassis can accommodate up to eight SCSI interface controllers, each of which has two SCSI I/O ports.

Chapter 14

The Graphics Display System

The Connection Machine graphics display system consists of two hardware components: a printed circuit module, called the *framebuffer*, which resides in the CM, and a high-resolution color monitor. Three coaxial cables connect the framebuffer to the color monitor. See Figure 12.

14.1 Connection Machine Framebuffer

The framebuffer resides on a backplane in one of the Connection Machine's 8K-processor sections, connecting directly to the 8K physical processors on that backplane via 256 I/O lines. The processors send data directly to the framebuffer over the I/O lines. The processors in other sections of the Connection Machine send data to the framebuffer indirectly, using the Connection Machine's interprocessor communications network to first send it to the 8K processors to which the framebuffer is directly connected.

The Color Buffers

The framebuffer contains a 7-megabyte display memory into which values from Connection Machine virtual processors are written. The display memory represents 2048 x 1024 pixels. The memory representing each pixel consists of three 8-bit color buffers — red, green, and blue — and one 4-bit overlay buffer.

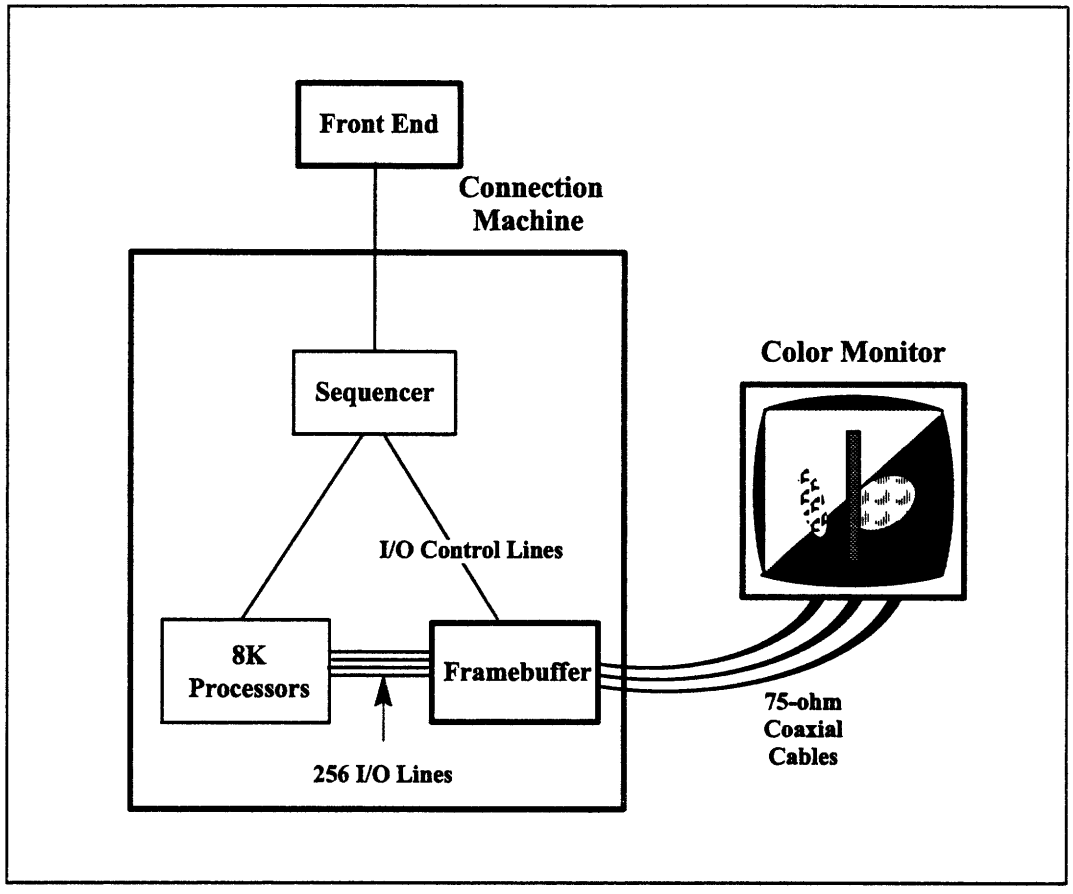


Figure 12. Connection Machine graphics system hardware components

Each of the 8-bit color buffers is associated with an 8-bit-wide color lookup table with 256 entries. The display memory may be configured to use either 24 bits per pixel or 8 bits per pixel. The number of bits per pixel is set when the display is initialized.

In *true-color* mode, 24 bits per pixel, each color buffer in display memory for each pixel is used as an index into the corresponding lookup table. The value at each entry in the tables determines the intensity of the red, green, and blue components of the color displayed for that pixel; 0 is off, 255 is full intensity.

In *pseudo-color* mode, 8 bits per pixel, only one of the color buffers in display memory, either green or blue, is read. This single value is used as the index into all three color tables to determine the color displayed.

Using pseudo-color mode limits the number of colors simultaneously available to 256. However, it requires only one-third as much Connection Machine memory and one-third as much data to be transferred to the framebuffer as does true-color mode.

Another advantage of using pseudo-color mode is the ability to use *double buffering*: using one buffer to hold the image currently shown while the next image is created in the other buffer. Switching buffers updates the image.

The Overlay Buffer

An *overlay buffer* is supported, which can be used to display text, cursors, and other data without disturbing the underlying image.

The overlay buffer is a 4-bit field maintained for each pixel. If the overlay is enabled, the value in the overlay buffer controls the display of each pixel as follows:

b3	b2	b1	b0		
0	0	0	0	–	the overlay for this pixel is off
0	0	0	1	–	display the color in the first overlay color register
0	0	1	0	–	display the color in the second overlay color register
0	0	1	1	–	display the color in the third overlay color register
0	1	x	x	–	override bits b0 and b1 to set pixel to white
1	x	x	x	–	override bits b0, b1, and b2 to set pixel to black

For each pixel, the lower two bits (b1 and b0) either turn off the overlay for the pixel, or select which register's color is displayed. Each register contains a 24-bit color value. The lowest eight bits control the red value, the middle bits control the green value, and the upper eight control the blue. These registers can be loaded at any time.

The upper two bits (b3 and b2) of the overlay planes are *override* bits. If b2 is set, then the pixel is set to a value that is 10 percent brighter than white ("whiter than white") regardless of bits b1 and b0. If b3 is set, then the pixel is set to a value of "blacker than black" (by 10 percent), regardless of b2, b1, and b0.

14.2 The Monitor

The framebuffer drives three 75-ohm coaxial cables connecting it to the color monitor with the red, green, and blue color values determined for each pixel. Currently the monitor is either a high-resolution Sony Trinitron Graphic Display Monitor or an NTSC-compatible color monitor.

The resolution of the monitor used determines the maximum size, in pixels, of the image displayed. The Sony monitor, like most high-resolution monitors, displays 1280 x 1024 pixels. NTSC-compatible video is displayed at a resolution of 640 x 512 pixels.